深入理解 TypeScript

书栈(BookStack.CN)

VISIT…

LANZAROTE Caliente.com

目　录

致谢
写在前面 - 深入理解 TypeScript
TypeScript 项目
- 编译上下文
- 声明空间
- 模块
- 命名空间
- 动态导入表达式

TypeScript 类型系统
- 概览
- 从 JavaScript 迁移
- @types
- 环境声明
- 接口
- 枚举
- lib.d.ts
- 函数
- 可调用的
- 类型断言
- Freshness
- 类型保护
- 字面量类型
- readonly
- 泛型
- 类型推断
- 类型兼容性
- Never
- 辨析联合类型
- 索引签名
- 流动的类型
- 异常处理
- 混合
- ThisType

JSX
- 支持 JSX
- React JSX

- 非 React JSX
TypeScript 错误提示
- 解读 Errors
- 常见的 Error
TIPs
- 基于字符串的枚举
- 名义化类型
- 状态函数
- Bind 是有害的
- 柯里化
- 泛型的实例化类型
- 对象字面量的惰性初始化
- 类是有用的
- export default 被认为是有害的
- 减少 setter 属性的使用
- 创建数组
- 谨慎使用 —outFile
- TypeScript 中的静态构造函数
- 单例模式
- 函数参数
- Truthy
- 构建切换
- 类型安全的 Event Emitter
- Reflect Metadata
- 协变与逆变
TypeScript 编译原理
- 概览
- 程序
- 抽象语法树
- 扫描器
- 解析器
- 绑定器
- 检查器
- 发射器

致谢

当前文档 《深入理解 TypeScript》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-01-22。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/TypeScriptDeepDiveZH

书栈官网：http://www.bookstack.cn

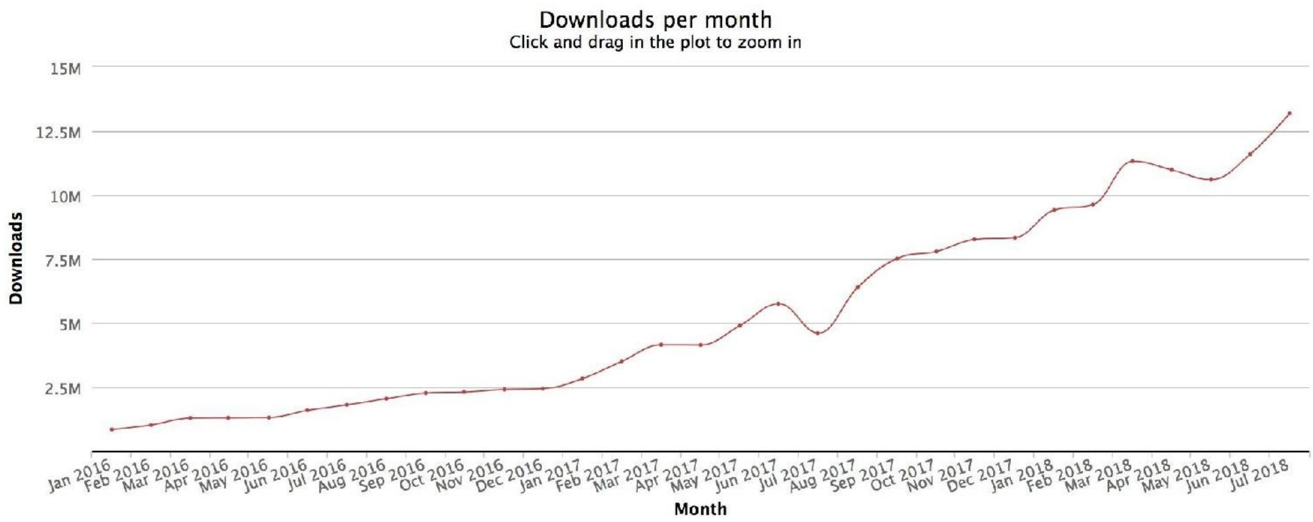
书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

深入理解 TypeScript

此书是 《TypeScript Deep Dive》 的中文翻译版，感谢作者 Basarat 的付出。

Why



> 数据来源：*npm* 包下载量

如你所见，TypeScript 发展至今，已经成为大型项目的标配，其提供的静态类型系统，大大增强了代码的可读性以及可维护性；同时，它提供最新和不断发展的 JavaScript 特性，能让我们建立更健壮的组件。

《TypeScript Deep Dive》 是一本很好的开源书，从基础到深入，很全面的阐述了 TypeScript 的各种魔法，不管你是新手，还是老鸟，它都将适应你。此外，它不同于 TypeScript 官方给出的文档（当然 TypeScript 给出的文档是很好的），在此书中，结合实际应用下的场景用例，你将能更深入的理解 TypeScript。

如今社区已经存在部分翻译，但都似乎已经停止更新。

于是在某天的某个冲动之下，这个 RP 就诞生了。

翻译内容

《TypeScript Deep Dive》 书中包含一部分 JavaScript Future 和一些其他的内容，在这里，我们并不打算翻译它，如果你有兴趣，可以查看原书中 JavaScript Future 的有关章节。

由于 TypeScript 更新频繁，在此书中，我也将加入一些原书中并没有涉及到的知识点，希望和大家相互学习，一起进步。

此外，在不违背原作者本意前提下，为了更直观的表达，部分内容将采用意译，而非直译。

How to contribute

你可以：

- 通过 PR 修改错别字，或者错误的格式；
- 发 issue 讨论文章中出现的一些不合理地方；
- 翻译 TODO 文件夹下的文章，并顺手 Email 我。
  希望你在翻译或者 PR 之前，阅读中文文章排版指北。

最后

如果你和我一样对 TypeScript 充满兴趣，可以订阅（star）本项目，及时收到有关于此项目的更新。

如果你对文章有任何疑问，欢迎提交 issues 和我交流。

如果你认为有些地方翻译不够准确，或者你想补充一些文中没提到但是非常有意思的知识点，欢迎 PR。

> 原文: *https://jkchao.github.io/typescript-book-chinese/*

- 编译上下文
- 声明空间
- 模块
- 命名空间
- 动态导入表达式

编译上下文

编译上下文算是一个比较花哨的术语，它用来给文件分组，告诉 TypeScript 哪些文件是有效的，哪些是无效的。除了有效文件所携带信息外，编译上下文也包含了有哪些编译选项正在使用。定义这种逻辑分组，一个比较好的方式是使用 `tsconfig.json` 文件。

tsconfig.json

基础

开始使用 `tsconfig.json` 是一件比较容易的事，你仅仅需要写下：

```
{}
```

例如在你项目的根目录下创建一个空 json 文件。通过此种方式，TypeScript 将会把此目录下（和子目录）的所有 `.ts` 文件做为编译上下文的一部分，它还会包含一部分默认的编译选项。

编译选项

你可以通过 `compilerOptions` 来定制你的编译选项：

```
{
  "compilerOptions": {

    /* 基本选项 */
    "target": "es5",                       // 指定 ECMAScript 目标版本: 'ES3' (default), 'ES5', 'ES2015', 'ES2016', 'ES2017', or 'ESNEXT'
    "module": "commonjs",                  // 指定使用模块: 'commonjs', 'amd', 'system', 'umd' or 'es2015'
    "lib": [],                             // 指定要包含在编译中的库文件
    "allowJs": true,                       // 允许编译 javascript 文件
    "checkJs": true,                       // 报告 javascript 文件中的错误
    "jsx": "preserve",                     // 指定 jsx 代码的生成: 'preserve', 'react-native', or 'react'
    "declaration": true,                   // 生成相应的 '.d.ts' 文件
    "sourceMap": true,                     // 生成相应的 '.map' 文件
    "outFile": "./",                       // 将输出文件合并为一个文件
    "outDir": "./",                        // 指定输出目录
    "rootDir": "./",                       // 用来控制输出目录结构 --outDir.
    "removeComments": true,                // 删除编译后的所有的注释
```

```
    "noEmit": true,                        // 不生成输出文件
    "importHelpers": true,                 // 从 tslib 导入辅助工具函数
    "isolatedModules": true,               // 将每个文件做为单独的模块 （与 'ts.transpileModule' 类似）.

    /* 严格的类型检查选项 */
    "strict": true,                        // 启用所有严格类型检查选项
    "noImplicitAny": true,                 // 在表达式和声明上有隐含的 any类型时报错
    "strictNullChecks": true,              // 启用严格的 null 检查
    "noImplicitThis": true,                // 当 this 表达式值为 any 类型的时候，生成一个错误
    "alwaysStrict": true,                  // 以严格模式检查每个模块，并在每个文件里加入 'use strict'

    /* 额外的检查 */
    "noUnusedLocals": true,                // 有未使用的变量时，抛出错误
    "noUnusedParameters": true,            // 有未使用的参数时，抛出错误
    "noImplicitReturns": true,             // 并不是所有函数里的代码都有返回值时，抛出错误
    "noFallthroughCasesInSwitch": true,    // 报告 switch 语句的 fallthrough 错误。（即，不允许 switch 的 case 语句贯穿）

    /* 模块解析选项 */
    "moduleResolution": "node",            // 选择模块解析策略： 'node' (Node.js) or 'classic' (TypeScript pre-1.6)
    "baseUrl": "./",                       // 用于解析非相对模块名称的基目录
    "paths": {},                           // 模块名到基于 baseUrl 的路径映射的列表
    "rootDirs": [],                        // 根文件夹列表，其组合内容表示项目运行时的结构内容
    "typeRoots": [],                       // 包含类型声明的文件列表
    "types": [],                           // 需要包含的类型声明文件名列表
    "allowSyntheticDefaultImports": true,  // 允许从没有设置默认导出的模块中默认导入。

    /* Source Map Options */
    "sourceRoot": "./",                    // 指定调试器应该找到 TypeScript 文件而不是源文件的位置
    "mapRoot": "./",                       // 指定调试器应该找到映射文件而不是生成文件的位置
    "inlineSourceMap": true,               // 生成单个 soucemaps 文件，而不是将 sourcemaps 生成不同的文件
    "inlineSources": true,                 // 将代码与 sourcemaps 生成到一个文件
```

```
    中，要求同时设置了 --inlineSourceMap 或 --sourceMap 属性

    /* 其他选项 */
    "experimentalDecorators": true,        // 启用装饰器
    "emitDecoratorMetadata": true          // 为装饰器提供元数据的支持
  }
}
```

这些（或者更多）编译选项，将会在稍后讨论。

TypeScript 编译

好的 IDE 支持对 TypeScript 的即时编译。但是，如果你想在使用 `tsconfig.json` 时从命令行手动运行 TypeScript 编译器，你可以通过以下方式：

- 运行 tsc，它会在当前目录或者是父级目录寻找 `tsconfig.json` 文件。
- 运行 `tsc -p ./path-to-project-directory` 。当然，这个路径可以是绝对路径，也可以是相对于当前目录的相对路径。
  你甚至可以使用 `tsc -w` 来启用 TypeScript 编译器的观测模式，在检测到文件改动之后，它将重新编译。

哪个文件？

你也可以显式指定需要编译的文件：

```
{
  "files": [
    "./some/file.ts"
  ]
}
```

或者，你可以使用 `include` 和 `exclude` 选项来指定需要包含的文件，和排除的文件：

```
{
  "include": [
    "./folder"
  ],
  "exclude": [
    "./folder/**/*.spec.ts",
    "./folder/someSubFolder"
  ]
```

```
9. }
```

注意

使用 `globs` ： `/*` （一个示例用法： `some/folder/` `/*` ）意味着匹配所有的文件夹和所有文件（扩展名为 `.ts/.tsx` ，当开启了 `allowJs: true` 选项时，扩展名可以是 `.js/.jsx` ）。

> 原文: *https://jkchao.github.io/typescript-book-chinese/project/compilationContext.html*

声明空间

在 TypeScript 里存在两种声明空间：类型声明空间与变量声明空间。我将会在下文中和大家讨论这两个概念。

类型声明空间

类型声明空间包含用来当做类型注解的内容，例如以下的一些类型声明：

```
class Foo {}
interface Bar {}
type Bas = {};
```

你可以将 `Foo`，`Bar`，`Bas` 做为类型注解使用，例如：

```
let foo: Foo;
let bar: Bar;
let bas: Bas;
```

注意，尽管你定义了 `interface Bar`，你并不能够将它做为一个变量使用，因为它没有定义在变量声明空间中：

```
interface Bar {}
const bar = Bar; // Error: "cannot find name 'Bar'"
```

提示 `cannot find name 'Bar'` 的原因是名称 `Bar` 并未定义在变量声明空间。这将带领我们进入下一个主题 "变量声明空间"。

变量声明空间

变量声明空间包含可用作变量的内容，在上文中 `Class Foo` 提供了一个类型 `Foo` 到类型声明空间，此外它同样提供了一个变量 `Foo` 到变量声明空间，如下所示：

```
class Foo {}
const someVar = Foo;
const someOtherVar = 123;
```

这很棒，尤其是当你想把一个类来当做变量传递时。

WARNING

我们并不能使用一些像 `interface` 定义的内容，来当做变量使用。

与此相似，一些像你用 `var` 声明的变量，也仅能在变量声明空间使用，不能用作类型注解。

```
const foo = 123;
let bar: foo; // ERROR: "cannot find name 'foo'"
```

提示 `cannot find name` 的原因是，名称 `foo` 没有定义在类型声明空间里。

> 原文: *https://jkchao.github.io/typescript-book-chinese/project/declarationspaces.html*

模块

全局模块

默认情况下，当你开始在一个新的 TypeScript 文件中写下代码时，它处于全局命名空间中。如在 `foo.ts` 里的以下代码：

```
const foo = 123;
```

如果你在相同的项目里创建了一个新的文件 `bar.ts`，TypeScript 类型系统将会允许你使用变量 `foo`，就好像它在全局可用一样：

```
const bar = foo; // allowed
```

毋庸置疑，使用全局变量空间是危险的，因为它会与文件内的代码命名冲突。我们推荐使用下文中将要提到的文件模块。

文件模块

它也被称为外部模块。如果在你的 TypeScript 文件的根级别位置含有 `import` 或者 `export`，它会在这个文件中创建一个本地的作用域。因此，我们需要把上文 `foo.ts` 改成如下方式（注意 `export` 用法）：

```
export const foo = 123;
```

在全局命名空间里，我们不再有 `foo`，这可以通过创建一个新文件 `bar.ts` 来证明：

```
const bar = foo; // ERROR: "cannot find name 'foo'"
```

如果你想在 `bar.ts` 里使用来自 `foo.ts` 的内容，你必须显式导入它，更新 `bar.ts` 如下所示：

```
import { foo } from './foo';
const bar = foo; // allow
```

在 `bar.ts` 文件里使用 `import`，不但允许你使用从其他文件导入的内容，而且它会将此文件 `bar.ts` 标记为一个模块，文件内定义的声明也不会污染全局命名空间。

文件模块详情

文件模块拥有强大的能力和可用性。在这里，我们来讨论它的能力以及一些用法。

澄清：commonjs, amd, es modules, others

首先，我们需要澄清这些模块系统的不一致性。我将会提供给你我当前的建议，以及消除一些顾虑。

你可以根据不同的 `module` 选项来把 TypeScript 编译成不同的 JavaScript 模块类型，这有一些你可以忽略的：

- AMD：不要使用它，它仅能在浏览器工作；
- SystemJS：这是一个好的实验，已经被 ES 模块替代；
- ES 模块：它并没有准备好。
  使用 `module: commonjs` 选项来替代这些模式，这会是一个好的主意。

怎么书写 TypeScript 模块，这也是一件让人困惑的事。在今天我们应该这么做：

- `import foo = require('foo')` 例如： `import/require` 使用 ES 模块语法。
  这很酷，接下来，让我们看看 ES 模块语法。

TIP

使用 `module: commonjs` 选项以及使用 ES 模块语法导入导出其他模块。

ES 模块语法

- 使用 `export` 关键字导出一个变量（或者类型）：

```
// foo.ts
export const someVar = 123;
export type someType = {
  foo: string;
};
```

- `export` 的写法除了上面这样，还有另外一种：

```
// foo.ts
const someVar = 123;
type someType = {
  type: string;
};

```

```
export { someVar, someType };
```

- 你也可以重命名变量导出：

```
// foo.ts
const someVar = 123;
export { someVar as aDifferentName };
```

- 使用 `import` 关键字导入一个变量或者是一个类型：

```
// bar.ts
import { someVar, someType } from './foo';
```

- 重命名导入变量或者类型：

```
// bar.ts
import { someVar as aDifferentName } from './foo';
```

- 除了指定加载某个输出值，还可以使用整体加载，即用星号（*）指定一个对象，所有输出值都加载在这个对象上面：

```
// bar.ts
import * as foo from './foo';
// 你可以使用 `foo.someVar` 和 `foo.someType` 以及其他任何从 `foo` 导出的变量或者类型
```

- 仅导入模块：

```
import 'core-js'; // 一个普通的 polyfill 库
```

- 从其他模块导入后整体导出：

```
export * from './foo';
```

- 从其他模块导入后，部分导出：

```
export { someVar } from './foo';
```

- 通过重命名，部分导出从另一个模块导入的项目：

```
export { someVar as aDifferentName } from './foo';
```

默认导入／导出

我并不喜欢用默认导出，虽然有默认导出的语法：

- 使用 `export default`
  - 在一个变量之前（不需要使用 `let/const/var`）；
  - 在一个函数之前；
  - 在一个类之前。

```
// some var
export default (someVar = 123);

// some function
export default function someFunction() {}

// some class
export default class someClass {}
```

- 导入使用 `import someName from 'someModule'` 语法（你可以根据需要为导入命名）：

```
import someLocalNameForThisFile from './foo';
```

模块路径

TIP

假设你使用 `moduleResolution: node` 选项。这个选项应该在你 TypeScript 配置文件里。如果你使用了 `module: commonjs` 选项， `moduleResolution: node` 将会默认开启。

这里存在两种不同截然不同的模块，它们是由导入语句中的不同的路径写法所引起的（例如：`import foo from 'THIS IS THE PATH SECTION'`）。

- 相对模块路径（路径以 `.` 开头，例如：`./someFile` 或者 `../../someFolder/someFile` 等）；
- 其他动态查找模块（如：`core-js`，`typestyle`，`react` 或者甚至是 `react/core` 等）。
  它们的主要区别来自于系统如何解析模块。

TIP

我将会使用一个概念性术语，`place` — 将在提及查找模式后解释它。

相对模块路径

这很简单，仅仅是按照相对路径：

- 如果文件 `bar.ts` 中含有 `import * as foo from './foo'`，`foo` 文件所存在的地方必须是相同文件夹下；
- 如果文件 `bar.ts` 中含有 `import * as foo from '../foo'`，`foo` 文件所存在的地方必须是上一级目录；
- 如果文件 `bar.ts` 中含有 `import * as foo from '../someFolder/foo'`，`foo` 文件所在的文件夹 `someFolder` 必须与 `bar.ts` 所在文件夹在相同的目录下。
  或者，你还可以想想其他相对路径导入的情景。？

动态查找

当导入路径不是相对路径时，模块解析将会模仿 Node 模块解析策略，以下我将给出一个简单例子：

- 当你使用 `import * as foo from 'foo'`，将会按如下顺序查找模块：
  - `./node_modules/foo`
  - `../node_modules/foo`
  - `../../node_modules/foo`
  - 直到系统的根目录
- 当你使用 `import * as foo from 'something/foo'`，将会按照如下顺序查找内容
  - `./node_modules/something/foo`
  - `../node_modules/something/foo`
  - `../../node_modules/something/foo`
  - 直到系统的根目录

什么是 `place`

当我提及被检查的 `place` 时，我想表达的是在这个 `place`，TypeScript 将会检查以下内容（例如一个 `foo` 的位置）：

- 如果这个 `place` 表示一个文件，如：`foo.ts`，欢呼！
- 否则，如果这个 `place` 是一个文件夹，并且存在一个文件 `foo/index.ts`，欢呼！
- 否则，如果这个 `place` 是一个文件夹，并且存在一个 `foo/package.json` 文件，在该文件中指定 `types` 的文件存在，那么就欢呼！
- 否则，如果这个 `place` 是一个文件夹，并且存在一个 `package.json` 文件，在该文件中指定 `main` 的文件存在，那么就欢呼！
  从文件类型上来说，我实际上是指 `.ts`， `.d.ts` 或者 `.js`

就是这样，现在你已经是一个模块查找专家（这并不是一个小小的成功）。

重写类型的动态查找

在你的项目里，你可以通过 `declare module 'somePath'` 来声明一个全局模块的方式，用来解决查找模块路径的问题：

```
// globals.d.ts
declare module 'foo' {
  // some variable declarations
  export var bar: number;
}
```

接着：

```
// anyOtherTsFileInYourProject.ts
import * as foo from 'foo';
// TypeScript 将假设（在没有做其他查找的情况下）
// foo 是 { bar: number }
```

import/require 仅仅是导入类型

以下导入语法：

```
import foo = require('foo');
```

它实际上只做了两件事：

- 导入 foo 模块的所有类型信息；
- 确定 foo 模块运行时的依赖关系。
  你可以选择仅加载类型信息，而没有运行时的依赖关系。在继续之前，你可能需要重新阅读本书的 声明空间部分 部分。

如果你没有把导入的名称当做变量声明空间来用，在编译成 JavaScript 时，导入的模块将会被完全移除。这有一些较好的例子，当你了解了它们之后，我们将会给出一些使用例子。

例子 1

```
import foo = require('foo');
```

将会编译成 JavaScript：

```

```

这是正确的，一个没有被使用的空文件。

例子 2

```
import foo = require('foo');
var bar: foo;
```

将会被编译成：

```
let bar;
```

这是因为 foo （或者其他任何属性如： `foo.bas` ）没有被当做一个变量使用。

例子 3

```
import foo = require('foo');
const bar = foo;
```

将会被编译成（假设是 commonjs）：

```
const foo = require('foo');
const bar = foo;
```

这是因为 `foo` 被当做变量使用了。

使用例子：懒加载

类型推断需要提前完成，这意味着，如果你想在 `bar` 文件里，使用从其他文件 `foo` 导出的类型，你将不得不这么做：

```
import foo = require('foo');
let bar: foo.SomeType;
```

然而，在某些情景下，你只想在需要时加载模块 `foo` ，此时你需要仅在类型注解中使用导入的模块名称，而不是在变量中使用。在编译成 JavaScript 式，这些将会被移除。接着，你可以手动导入你需要的模块。

做为一个例子，考虑以下基于 `commonjs` 的代码，我们仅在一个函数内导入 `foo` 模块：

```
import foo = require('foo');

export function loadFoo() {
  // 这是懒加载 foo，原始的加载仅仅用来做类型注解
```

```
  const _foo: typeof foo = require('foo');
  // 现在，你可以使用 `_foo` 替代 `foo` 来做为一个变量使用
}
```

一个同样简单的 `amd` 模块（使用 requirejs）：

```
import foo = require('foo');

export function loadFoo() {
  // 这是懒加载 foo，原始的加载仅仅用来做类型注解
  require(['foo'], (_foo: typeof foo) => {
    // 现在，你可以使用 `_foo` 替代 `foo` 来做为一个变量使用
  });
}
```

这些通常在以下情景使用：

- 在 web app 里， 当你在特定路由上加载 JavaScript 时；
- 在 node 应用里，当你只想加载特定模块，用来加快启动速度时。

使用例子：打破循环依赖

类似于懒加载的使用用例，某些模块加载器（commonjs/node 和 amd/requirejs）不能很好的处理循环依赖。在这种情况下，一方面我们使用延迟加载代码，并在另一方面预先加载模块时很实用的。

使用例子：确保导入

当你加载一个模块，只是想引入其附加的作用（如：模块可能会注册一些像 CodeMirror addons）时，然而，如果你仅仅是 `import/require` （导入）一些并没有与你的模块或者模块加载器有任何依赖的 JavaScript 代码，（如：webpack），经过 TypeScript 编译后，这些将会被完全忽视。在这种情况下，你可以使用一个 `ensureImport` 变量，来确保编译的 JavaScript 依赖与模块。如：

```
import foo = require('./foo');
import bar = require('./bar');
import bas = require('./bas');

const ensureImport: any = foo || bar || bas;
```

globals.d.ts

在上文中，当我们讨论文件模块时，比较了全局变量与文件模块，并且我们推荐使用基于文件的模块，而不是选择污染全局命名空间。

然而，如果你的团队里有 TypeScript 初学者，你可以提供他们一个 `globals.d.ts` 文件，用来将一些接口或者类型放入全局命名空间里，这些定义的接口和类型能在你的所有 TypeScript 代码里使用。

TIP

对于任何需要编译成 JavaScript 代码，我们强烈建议你放入文件模块里。

- `globals.d.ts` 是一种扩充 `lib.d.ts` 很好的方式，如果你需要。
- 当你从 `TS` 迁移到 `JS` 时，定义 `declare module "some-library-you-dont-care-to-get-defs-for"` 能让你快速开始。

> 原文: *https://jkchao.github.io/typescript-book-chinese/project/modules.html*

命名空间

在 JavaScript 使用命名空间时， 这有一个常用方便的语法：

```
(function(something) {
  something.foo = 123;
})(something || (something = {}));
```

`something || (something = {})` 允许匿名函数 `function (something) {}` 添加属性至已经存在的对象上，或者会创建一个新对象，然后添加属性至新对象上，这意味着你可以拥有由某些边界拆成的不同的块：

```
(function(something) {
  something.foo = 123;
})(something || (something = {}));

console.log(something);
// { foo: 123 }

(function(something) {
  something.bar = 456;
})(something || (something = {}));

console.log(something); // { foo: 123, bar: 456 }
```

在确保创建的变量不会泄漏至全局变量中时，这种方式在 JavaScript 中很常见。当使用基于文件模块时，你无须担心这点，但是此种方式，仍然适用于合理的函数逻辑分组中。因此 TypeScript 提供了 `namespace` 关键字用来描述这种分组，如下所示：

```
namespace Utility {
  export function log(msg) {
    console.log(msg);
  }
  export function error(msg) {
    console.log(msg);
  }
}

// usage
Utility.log('Call me');
```

```
Utility.error('maybe');
```

`namespace` 关键字通过 TypeScript 编译后，与我们看到的 JavaScript 代码一样：

```
(function (Utility) {
  // 添加属性至 Utility
})(Utility || Utility = {});
```

有一点值得注意的是，命名空间是支持嵌套的。因此，你可以做一些类似于在 `Utility` 命名空间下嵌套一个命名空间 `Messaging` 的事情。

对大多数项目来说，我们推荐使用一个使用 `namespace` 的外部的模块，用来快速的演示和移植旧的 JavaScript 代码。

> 原文: *https://jkchao.github.io/typescript-book-chinese/project/namespaces.html*

动态导入表达式

动态导入表达式是 ECMASript 的一个新功能，它允许你在你程序的任意地方异步加载一个模块，TC39 JavaScript 委员会有这样一个处于 stage 3 的一个提案，它被称为 import() proposal for JavaScript。

此外，**webpack** bundler 有一个 `Code Splitting` 的功能，它能允许你将代码拆分为许多块，这些块在将来可被异步下载。因此，你可以在程序中首先提供一个最小的程序启动包，并在将来异步加载其他模块。

这很自然的想到（如果我们工作在 webpack dev 的工作流程）TypeScript 2.4 dynamic import expressions 将会把你最终生成的 JavaScript 代码自动分割成很多块。但是这似乎并不容易实现，因为它依赖于我们正在使用的 `tsconfig.json` 配置文件。

webpack 实现代码分割的方式有两种：使用 `import()` （首选，ECMAScript 的提案）和 `require.ensure()` （最后考虑，webpack 具体实现）。因此，我们期望 TypeScript 的输出是保留 `import()` 语句，而不是将其转化为其他任何代码。

让我们来看一个例子，在这个例子中，我们演示了如何配置 webpack 和 TypeScript 2.4 +。

如下例子，我希望懒加载 `moment` 库，同时我也希望使用代码分割的功能，这意味 `moment` 会被分割到一个单独的 JavaScript 文件，当它被使用时，会被异步加载。

```
import(/* webpackChunkName: "momentjs" */ 'moment')
  .then(moment => {
    // 懒加载的模块拥有所有的类型，并且能够按期工作
    // 类型检查会工作，代码引用也会工作  :100:
    const time = moment().format();
    console.log('TypeScript >= 2.4.0 Dynamic Import Expression:');
    console.log(time);
  })
  .catch(err => {
    console.log('Failed to load moment', err);
  });
```

这是 `tsconfig.json` 的配置文件：

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "esnext",
```

```
    "lib": [
      "dom",
      "es5",
      "scripthost",
      "es2015.promise"
    ],
    "jsx": "react",
    "declaration": false,
    "sourceMap": true,
    "outDir": "./dist/js",
    "strict": true,
    "moduleResolution": "node",
    "typeRoots": [
      "./node_modules/@types"
    ],
    "types": [
      "node",
      "react",
      "react-dom"
    ]
  }
}
```

重要的提示

- 使用 "module": "esnext" 选项：TypeScript 为 Webpack Code Splitting 生成 `import()` 语句。
- 进一步了解有关信息，推荐阅读这篇文章：Dynamic Import Expressions and webpack 2 Code Splitting integration with TypeScript 2.4.

> 原文: *https://jkchao.github.io/typescript-book-chinese/project/dynamicImportExpressions.html*

- 概览
- 从 JavaScript 迁移
- @types
- 环境声明
- 接口
- 枚举
- lib.d.ts
- 函数
- 可调用的
- 类型断言
- Freshness
- 类型保护
- 字面量类型
- readonly
- 泛型
- 类型推断
- 类型兼容性
- Never
- 辨析联合类型
- 索引签名
- 流动的类型
- 异常处理
- 混合
- ThisType

概览

TypeScript 类型系统

在讨论为什么使用 TypeScript 时，我们涵盖了 TypeScript 类型系统的主要功能。以下是讨论的一些关键点：

- TypeScript 类型系统设计是可选的，因此，你的 JavaScript 即是 TypeScript;
- TypeScript 不会阻止 JavaScript 的运行，即使存在类型错误也不例外，这能让你的 JavaScript 逐步迁移至 TypeScript。
  现在让我们开始学习 TypeScript 类型系统的语法，在这一章节中，你将能给你的代码加上类型注解，并且能看到它的益处。这将为我们进一步了解类型系统做铺垫。

基本注解

如前文所提及，类型注解使用 `:TypeAnnotation` 语法。类型声明空间中可用的任何内容都可以用作类型注解。

在下面这个例子中，使用了变量、函数参数以及函数返回值的类型注解：

```
const num: number = 123;
function identity(num: number): number {
  return num;
}
```

原始类型

JavaScript 原始类型也同样适应于 TypeScript 的类型系统，因此 `string` 、 `number` 、 `boolean` 也可以被用作类型注解：

```
let num: number;
let str: string;
let bool: boolean;

num = 123;
num = 123.456;
num = '123'; // Error

```

```
str = '123';
str = 123; // Error

bool = true;
bool = false;
bool = 'false'; // Error
```

数组

TypeScript 为数组提供了专用的类型语法，因此你可以很轻易的注解数组。它使用后缀 `[]` ，接着你可以根据需要补充任何有效的类型注解（如： `:boolean[]` ）。它能让你安全的使用任何有关数组的操作，而且它也能防止一些类似于赋值错误类型给成员的行为。如下所示：

```
const boolArray: boolean[];

boolArray = [true, false];
console.log(boolArray[0]); // true
console.log(boolArray.length); // 2

boolArray[1] = true;
boolArray = [false, false];

boolArray[0] = 'false'; // Error
boolArray = 'false'; // Error
boolArray = [true, 'false']; // Error
```

接口

接口是 TypeScript 的一个核心知识，它能合并众多类型声明至一个类型声明：

```
interface Name {
  first: string;
  second: string;
}

let name: Name;
name = {
  first: 'John',
  second: 'Doe'
};
```

```

name = {
  // Error: 'Second is missing'
  first: 'John'
};

name = {
  // Error: 'Second is the wrong type'
  first: 'John',
  second: 1337
};
```

在这里，我们把类型注解：`first: string` + `second: string` 合并到了一个新的类型注解里 `Name` ，这样能强制对每个成员进行类型检查。接口在 TypeScript 拥有强大的力量，在稍后，我们将会用整个章节来阐述如何更好的使用它。

内联类型注解

与创建一个接口不同，你可以使用内联注解语法注解任何内容：`:{ /Structure/ }` ：

```
let name: {
  first: string;
  second: string;
};

name = {
  first: 'John',
  second: 'Doe'
};

name = {
  // Error: 'Second is missing'
  first: 'John'
};

name = {
  // Error: 'Second is the wrong type'
  first: 'John',
  second: 1337
};
```

内联类型能为你快速的提供一个类型注解。它可以帮助你省去为类型起名的麻烦（你可能会使用一个很糟糕的名称）。然而，如果你发现需要多次使用相同的内联注解时，考虑把它重构为一个接口（或者是 `type alias` ，它会在接下来的部分提到）是一个不错的注意。

特殊类型

除了被提到的一些原始类型，在 TypeScript 中，还存在一些特殊的类型，它们是 `any` 、 `null` 、 `undefined` 以及 `void` 。

any

`any` 类型在 TypeScript 类型系统中占有特殊的地位。它提供给你一个类型系统的「后门」，TypeScript 将会把类型检查关闭。在类型系统里 `any` 能够兼容所有的类型（包括它自己）。因此，所有类型都能被赋值给它，它也能被赋值给其他任何类型。以下有一个证明例子：

```
let power: any;

// 赋值任意类型
power = '123';
power = 123;

// 它也兼容任何类型
let num: number;
power = num;
num = power;
```

当你从 JavaScript 迁移至 TypeScript 时，你将会经常性使用 `any` 。但你必须减少对它的依赖，因为你需要确保类型安全。当使用 `any` 时，你基本上是在告诉 TypeScript 编辑器不要进行任何的类型检查。

null 和 undefined

在类型系统中，JavaScript 中的 null 和 undefined 字面量和其他被标注了 `any` 类型的变量一样，都能被赋值给任意类型的变量，如下例子所示：

```
let num: number;
let str: string;

// 这些类型能被赋予
num = null;
str = undefined;
```

void

使用 `:void` 来表示一个函数没有一个返回值

```
function log(message: string): void {
  console.log(message);
}
```

泛型

在计算机科学中，许多算法和数据结构并不会依赖于对象的实际类型。然而，你仍然会想在每个变量里强制提供约束。例如：在一个函数中，它接受一个列表，并且返回这个列表的反向排序，这里的约束是指传入至函数的参数与函数的返回值：

```
function reverse<T>(items: T[]): T[] {
  const toreturn = [];
  for (let i = items.length - 1; i >= 0; i--) {
    toreturn.push(items[i]);
  }
  return toreturn;
}

const sample = [1, 2, 3];
const reversed = reverse(sample);

console.log(reversed); // 3, 2, 1

// Safety
reversed[0] = '1'; // Error
reversed = ['1', '2']; // Error

reversed[0] = 1; // ok
reversed = [1, 2]; // ok
```

在上个例子中，函数 `reverse` 接受一个类型为 `T` （注意在 `reverse<T>` 中的类型参数）的数组（ `items: T[]` ），返回值为类型 T 的一个数组（注意：T[]），函数 `reverse` 的返回值类型与它接受的参数的类型一样。当你传入 `var sample = [1, 2, 3]` 时，TypeScript 能推断出 `reverse` 为 `number[]` 类型，从而能给你类型安全。于此相似，当你传入一个类型为 `string[]` 类型的数组时，TypeScrip 能推断 `reverse` 为 `string` [] 类型，如下例子所示：

```
const strArr = ['1', '2'];
let reversedStrs = reverse(strArr);

reversedStrs = [1, 2]; // Error
```

事实上，JavaScript 数组已经拥有了 `reverse` 的方法，TypeScript 也确实使用了泛型来定义其结构：

```
interface Array<T> {
  reverse(): T[];
}
```

这意味着，当你在数组上调用 `.reverse` 方法时，得会获得类型安全：

```
let numArr = [1, 2];
let reversedNums = numArr.reverse();

reversedNums = ['1', '2']; // Error
```

当在章节 环境声明 中提及了 `lib.d.ts` 时，我们会讨论更多关于 `Array<T>` 的信息。

联合类型

在 JavaScript 中，你希望属性为多种类型之一，如字符串或者数组。这就是联合类型所能派上用场的地方（它使用 `|` 作为标记，如 `string | number` ）。在函数参数里。一个常见的用例是一个可以接受单个对象或对象数组的函数：

```
function formatCommandline(command: string[] | string) {
  let line = '';
  if (typeof command === 'string') {
    line = command.trim();
  } else {
    line = command.join(' ').trim();
  }

  // Do stuff with line: string
}
```

交叉类型

在 JavaScript 中， `extend` 是一种非常常见的模式，在这种模式中，你可以从两个对象中创建一个新对象，新对象会拥有着两个对象所有的功能。交叉类型可以让你安全的使用此种模式：

```
function extend<T, U>(first: T, second: U): T & U {
  const result = <T & U>{};
  for (let id in first) {
    (<T>result)[id] = first[id];
  }
  for (let id in second) {
    if (!result.hasOwnProperty(id)) {
      (<U>result)[id] = second[id];
    }
  }

  return result;
}

const x = extend({ a: 'hello' }, { b: 42 });

// 现在 x 拥有了 a 属性与 b 属性
const a = x.a;
const b = x.b;
```

元组类型

JavaScript 并没有支持类似于元组的支持。开发者通常只能使用数组来表示元组，但是 TypeScript 类型系统支持它。使用 `:[typeofmember1, typeofmember2]` 能够为元组添加类型注解，元组可以包含任意数量的成员，以下例子演示了元组：

```
let nameNumber: [string, number];

// Ok
nameNumber = ['Jenny', 221345];

// Error
nameNumber = ['Jenny', '221345'];
```

将其与 TypeScript 中的解构一起使用：

```
let nameNumber: [string, number];
nameNumber = ['Jenny', 322134];
```

```
const [name, num] = nameNumber;
```

类型别名

TypeScript 提供使用类型注解的便捷语法，你可以使用 `type SomeName = someValidTypeAnnotation` 的语法来创建别名：

```
type StrOrNum = string | number;

// 使用
let sample: StrOrNum;
sample = 123;
sample = '123';

// 会检查类型
sample = true; // Error
```

不同于 `interface` 你可以提供一个类型别名至任意的类型注解上（在联合类型和交叉类型中比较实用），这有一些能让你熟悉语法的实例：

```
type Text = string | { text: string };
type Coordinates = [number, number];
type Callback = (data: string) => void;
```

TIP

- 如果你需要使用类型注解的层次结构，请使用接口。它能使用 `implements` 和 `extends`
- 为一个简单的对象类型（像例子中的 `Coordinates`）使用类型别名，仅仅有一个语义化的作用。与此相似，当你想给一个联合类型和交叉类型使用一个语意化的名称时，一个类型别名将会是一个好的选择。

最后

现在你已经能够为你的大部分 JavaScript 代码添加类型注解，接着，让我们深入了解 TypeScript 的类型系统吧。

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/overview.html*

从 JavaScript 迁移

假设：

- 你知道 JavaScript；
- 你知道在项目中使用常用的方式和构建工具（如：webpack）。
  有了以上假设，从 JavaScript 迁移，总的来说包括一下步骤：
- 添加一个 `tsconfig.json` 文件；
- 把文件扩展名从 `.js` 改成 `.ts`，开始使用 `any` 来减少错误；
- 开始在 TypeScript 中写代码，尽可能的减少 `any` 的使用；
- 回到旧代码，开始添加类型注解，并修复已识别的错误；
- 为你的第三方 JavaScript 代码定义环境声明。
  让我们进一步讨论其中的某几点。

记住所有的 JavaScript 都是有效的 TypeScript。这意味着，如果让 TypeScript 编译器编译 TypeScript 里的 JavaScript 代码，编译后的结果将会与原始的 JavaScript 代码一摸一样。也就是说，把文件扩展名从 `.js` 改成 `.ts` 将不会造成任何负面的影响。

减少错误

TypeScript 将会立即对你的代码进行类型检查，你的 JavaScript 代码可能并不像你想象中那样整齐了，因此你会收到一些报错信息。你可以使用 `any` 类型来解决它们中的大部分报错：

```
let foo = 123;
let bar = 'hey';

bar = foo; // Error: 不能把 number 类型赋值给 string 类型
```

虽然这些错误是有效的（并且在大多数情景下，推断信息的错误提示将比代码库的不同部分的原始作者想象的更好），但是你的重点可能是逐步迁移至 TypeScript，在这里，你可以使用类型断言来减少此错误：

```
let foo = 123;
let bar = 'hey';

bar = foo as any; // ok
```

在另一方面来说，你可能想用 `any` 用作类型注解：

```
function foo() {
  return 1;
}

let bar = 'hey';
bar = foo(); // Error: 不能把一个 number 类型赋值给 string 类型
```

抑制这种错误：

```
function foo(): any {
  // 添加 'any'
  return 1;
}

let bar = 'hey';
bar = foo();
```

NOTICE

使用此种方式来减少错误是危险的，但是它允许你将注意力转移到你的新 TypeScript 代码错误上。当你离开一段代码时，你可能会留下 `// TODO` 的注释。

第三方代码

你可以将你的 JavaScript 的代码改成 TypeScript 代码，但是你不能让这个世界都使用 TypeScript。这正是 TypeScript 环境声明支持的地方。我们建议你创建一个 `vendor.d.ts` 文件作为开始（ `.d.ts` 文件扩展名指定这个文件是一个声明文件），然后我们可以向文件里添加东西。或者，你也可以创建一个针对于特定库的声明文件，如为 jquery 创建 `jquery.d.ts` 文件。

NOTICE

几乎排名前 90% 的 JavaScript 库的声明文件存在于 DefinitelyTyped 这样一个仓库里，在创建自己定义的声明文件之前，我们建议你先去仓库中寻找。虽然创建一个声明文件这种快速但是不好的方式是减小使用 TypeScript 初始阻力的重要步骤。

考虑使用 `jquery` 的用例，你可以非常简单快速的为它创建一个定义：

```
declare var $: any;
```

有时候，你可能想给某些变量一些明确的定义（如： jquery ），并且你会在类型声明空间中使用它。你可以通过 `type` 关键字快速的实现它：

```
declare type JQuery = any;
declare var $: JQuery;
```

这提供给你一个更清晰的使用模式。

再一次说明，一个高质量的 `jquery.d.ts` 已经在 DefinitelyTyped 中存在。现在你已经知道当你使用 JavaScript 第三方模块时， 如何克服从 JavaScript 至 TypeScript 的阻力。在接下去的章节，我们将会讨论环境声明。

第三方的 NPM 模块

与全局变量声明相似，你可以快速的定义一个全局模块，如：对于 `jquery` ，如果你想把它作为一个模块来使用（NPM ），你可以自己通过以下方式实现：

```
declare module 'jquery';
```

然后你就可以在必要时导入它：

```
import * as $ from 'jquery';
```

TIP

再一次说明，一个高质量的 `jquery.d.ts` 已经在 DefinitelyTyped 中存在，但是可能在你的包里没有，所以，你现在有一个简单快速的方式来继续迁移。

额外的非 JavaScript 资源

在 TypeScript 中，甚至可以允许你导入任何文件，例如 `.css` 文件（如果你使用的是 webpack 样式加载器或 css 模块），你只要添加如下代码（放在 `globals.d.ts` ）：

```
declare module '*.css';
```

现在你可以使用 `import * as foo from './some/file.css'` 。

与此相似，如果你想使用 html 模版（例如：angular），你可以：

```
declare module '*.html';
```

> **原文**: *https://jkchao.github.io/typescript-book-chinese/typings/migrating.html*

@types

毫无疑问，DefinitelyTyped 是 TypeScript 最大的优势之一，社区已经记录了 90% 的顶级 JavaScript 库。

这意味着，你可以非常高效的使用这些库，而无需在单独的窗口打开相应文档（以确保输入的正确性）。

使用 @types

你可以通过 `npm` 来安装使用 `@types`，如下例所示，你可以为 `jquery` 添加声明文件：

```
npm install @types/jquery --save-dev
```

`@types` 支持全局和模块类型定义。

全局 @types

默认情况下，TypeScript 会自动包含支持全局使用的任何定义。例如，对于 jquery，你应该能够在项目中开始全局使用 `$`。

模块 @types

安装完之后，不需要特别的配置，你就可以像使用模块一样使用它：

```
import * as $ from 'jquery';

// 现在你可以此模块中任意使用$了 :)
```

控制全局

可以看出，对于某些团队而言，拥有允许全局泄漏的定义可能是一个问题。因此，你可以通过配置 `tsconfig.json` 的 `compilerOptions.types` 选项，引入有意义的类型：

```
{
  "compilerOptions": {
    "types" : [
      "jquery"
    ]
```

```
   }
}
```

如上例所示，通过配置 `compilerOptions.types: [ "jquery" ]` 后，只允许使用 `jquery` 的 `@types` 包，即使这个人安装了另一个声明文件，比如 `npm install @types/node`，它的全局变量（例如 `process`）也不会泄漏到你的代码中，直到你将它们添加到 tsconfig.json 类型选项。

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/types.html*

环境声明

正如我们在为什么使用 TypeScript 所说：

> *TypeScript* 的设计目标之一是让你从现有的 *JavaScript* 库中安全、轻松的使用 *TypeScript*，你可以通过 *TypeScript* 声明文件来做到这一点。

环境声明允许你安全的使用现有的 JavaScript 库，并且能让你的 JavaScript，CoffeeScript 或者其他需要编译成 JavaScript 的语言逐步迁移至 TypeScript。

学习为第三方 JavaScript 库编写环境声明，是一种为 TypeScript 写注解比较好的实践方式。

声明文件

你可以通过 `declare` 关键字，来告诉 TypeScript，你正在试图表述一个其他地方已经存在的代码（如：写在 JavaScript、CoffeeScript 或者是像浏览器和 Node.js 运行环境里的代码）：

```
foo = 123; // Error: 'foo' is not defined
```

和：

```
declare var foo: any;
foo = 123; // allow
```

你可以选择把这些声明放入 `.ts` 或者 `.d.ts` 里。在你实际的项目里，我们强烈建议你应该把声明放入 `.d.ts` 里（可以从一个命名为 `globals.d.ts` 或者 `vendor.d.ts` 文件开始）。

如果一个文件有扩展名 `.d.ts`，这意味着每个顶级的声明都必须以 `declare` 关键字作为前缀。这有利于向作者说明，在这里 TypeScript 将不会把它编译成任何代码，同时他需要确保这些在编译时存在。

TIP

- 环境声明就好像你与编译器之间的一个约定，如果这些没有在编译时存在，但是你却使用了它们，则事情将会在没有警告的情况下中断。
- 环境声明就好像是一个文档。如果源文件更新了，你应该同步更进。所以，当你使用源文件在运行时的新行为时，如果没有人更新环境声明，编译器将会报错。

变量

举个例子，当你想告诉 TypeScript 编辑器关于 `process` 变量时，你可以这么做：

```
declare let process: any;
```

TIP

你并不需要为 `process` 做这些，因为这已经存在于社区维护的 `node.d.ts`

这允许你使用 `process`，并能成功通过 TypeScript 的编译：

```
process.exit();
```

我们推荐尽可能的使用接口，例如：

```
interface Process {
  exit(code?: number): void;
}

declare let process: Process;
```

因为这允许其他人扩充这些全局变量，并且会告诉 TypeScript 有关于这些的修改。例如：考虑到以下情况，我们添加一个 `exitWithLogging` 函数至 `process`：

```
interface Process {
  exitWithLogging(code?: number): void;
}

process.exitWithLogging = function() {
  console.log('exiting');
  process.exit.apply(process, arguments);
};
```

接下来，让我们更详细的了解接口。

> 原文：*https://jkchao.github.io/typescript-book-chinese/typings/ambient.html*

接口

接口运行时的影响为 0。TypeScript 接口中有很多方式来声明变量的结构。

以下两个是等效声明, 第一个使用内联注解，第二个使用接口：

```
// Sample A
declare const myPoint: { x: number; y: number };

// Sample B
interface Point {
  x: number;
  y: number;
}
declare const myPoint: Point;
```

示例 B 的好处在于，如果有人创建了一个基于 `myPoint` 的库来添加新成员, 他们可以轻松将此成员添加到 `myPoint` 的现有声明中：

```
// Lib a.d.ts
interface Point {
  x: number,
  y: number
}
declare const myPoint: Point

// Lib b.d.ts
interface Point {
  z: number
}

// Your code
let myPoint.z // Allowed!
```

因为 TypeScript 接口是开放式的，这是 TypeScript 的一个重要原则，它允许你使用接口模仿 JavaScript 的可扩展性。

类可以实现接口

如果你希望在类中使用必须遵循的接口（类）或是别人定义的对象结构，可以使用 `implements` 关

键字来确保兼容性：

```
interface Point {
  x: number;
  y: number;
}

class MyPoint implements Point {
  x: number;
  y: number; // Same as Point
}
```

基本上在 `implements（实现）` 的存在下，该外部 `Point` 接口的任何更改都将导致代码库中的编译错误，因此可以轻松地使其保持同步：

```
interface Point {
  x: number;
  y: number;
  z: number; // New member
}

class MyPoint implements Point {
  // ERROR : missing member `z`
  x: number;
  y: number;
}
```

注意， `implements` 限制了类实例的结构，即：

```
let foo: Point = new MyPoint();
```

像 `foo: Point = MyPoint` 这样并不是一回事。

注意

并非每个接口都是很容易实现的

接口旨在声明 JavaScript 中可能存在的任意结构。

思考以下例子，其中可以使用 `new` 调用某些内容：

```
interface Crazy {
  new (): {
    hello: number;
  };
}
```

你可能会有这样的代码：

```
class CrazyClass implements Crazy {
  constructor() {
    return { hello: 123 };
  }
}

// Because
const crazy = new CrazyClass(); // crazy would be { hello:123 }
```

你可以使用接口声明所有的 JavaScript，甚至可以安全地从 TypeScript 中使用它们。但并不意味着你可以使用 TypeScript 类来实现它们。

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/interfaces.html*

枚举

枚举是组织收集有关联变量的一种方式，许多程序语言（如：c/c#/Java）都有枚举数据类型。下面是定义一个 TypeScript 枚举类型的方式：

```
enum CardSuit {
  Clubs,
  Diamonds,
  Hearts,
  Spades
}

// 简单的使用枚举类型
let Card = CardSuit.Clubs;

// 类型安全
Card = 'not a member of card suit'; // Error: string 不能赋值给 `CardSuit` 类型
```

这些枚举类型的值都是数字类型，因此它们被称为数字类型枚举。

数字类型枚举与数字类型

数字类型枚举，允许我们将数字类型或者其他任何与数字类型兼容的类型赋值给枚举类型的实例。

```
enum Color {
  Red,
  Green,
  Blue
}

let col = Color.Red;
col = 0; // 有效的，这也是 Color.Red
```

数字类型枚举与字符串类型

在我们继续深入枚举类型之前，先让我们看看它编译的 JavaScript，以下是一个简单的 TypeScript 枚举类型：

```
enum Tristate {
```

```
  False,
  True,
  Unknow
}
```

编译成 JavaScript:

```
var Tristate;
(function(Tristate) {
  Tristate[(Tristate['False'] = 0)] = 'False';
  Tristate[(Tristate['True'] = 1)] = 'True';
  Tristate[(Tristate['Unknown'] = 2)] = 'Unknown';
})(Tristate || (Tristate = {}));
```

先让我们聚焦于代码高亮的一行，其中 `Tristate['False'] = 0` 用于自我解释，例如将 `Tristate` 对象里的 `False` 成员值设置为 `0` 。注意，JavaScript 赋值运算符返回的值是被赋予的值（在此例子中是 `0` ），因此下一次 JavaScript 运行时执行的代码是 `Tristate[0] = 'False'` 。意味着你可以使用 `Tristate` 变量来把字符串枚举类型改造成一个数字或者是数字类型的枚举类型，如下所示：

```
enum Tristate {
  False,
  True,
  Unkown
}

console.log(Tristate[0]); // 'False'
console.log(Tristate['False']); // 0
console.log(Tristate[Tristate.False]); // 'False' because `Tristate.False == 0`
```

改变与数字枚举关联的数字

默认情况下，第一个枚举值是 `0` ，然后每个后续值依次递增 1：

```
enum Color {
  Red, // 0
  Green, // 1
  Blue // 2
}
```

但是，你可以通过特定的赋值来改变给任何枚举成员关联的数字，如下例子，我们从 3 开始依次递增：

```
enum Color {
  DarkRed = 3, // 3
  DarkGreen, // 4
  DarkBlue // 5
}
```

TIP

我通常用 `= 1` 初始化，因为在枚举类型值里，它能让你做一个安全可靠的检查。

使用数字类型作为标志

枚举的一个很好用途是使用枚举作为标志。这些标志允许你检查一组条件中的某个条件是否为真。考虑如下代码例子，我们有一组关于 animals 的属性：

```
enum AnimalFlags {
  None        = 0,
  HasClaws    = 1 << 0,
  CanFly      = 1 << 1,
  EatsFish    = 1 << 2,
  Endangered  = 1 << 3
}
```

在这里，我们使用了左移的位运算符，将数字 `1` 的二进制向左移动位置得到数字 `0001` 、 `0010` 、 `0100` 和 `1000`（换成十进制结果是：1, 2, 4, 8）。当你在使用这边标记的时候，这些位运算符 `|` （或）、 `&` （和）、 `~` （非）将会是你最好的朋友：

```
enum AnimalFlags {
  None        = 0,
  HasClaws    = 1 << 0,
  CanFly      = 1 << 1
}

interface Animal {
  flags: AnimalFlags;
  [key: string]: any;
}

```

```
function printAnimalAbilities(animal: Animal) {
  var animalFlags = animal.flags;
  if (animalFlags & AnimalFlags.HasClaws) {
    console.log('animal has claws');
  }
  if (animalFlags & AnimalFlags.CanFly) {
    console.log('animal can fly');
  }
  if (animalFlags == AnimalFlags.None) {
    console.log('nothing');
  }
}

var animal = { flags: AnimalFlags.None };
printAnimalAbilities(animal); // nothing
animal.flags |= AnimalFlags.HasClaws;
printAnimalAbilities(animal); // animal has claws
animal.flags &= ~AnimalFlags.HasClaws;
printAnimalAbilities(animal); // nothing
animal.flags |= AnimalFlags.HasClaws | AnimalFlags.CanFly;
printAnimalAbilities(animal); // animal has claws, animal can fly
```

在这里：

- 我们使用 `|=` 来添加一个标志；
- 组合使用 `&=` 和 `~` 来清理一个标志；
- `|` 来合并标志。

TIP

你可以组合标志，用来在枚举类型中定义方便快捷的方式，如下 `EndangeredFlyingClawedFishEating`：

```
enum AnimalFlags {
  None         = 0,
  HasClaws     = 1 << 0,
  CanFly       = 1 << 1,

  EndangeredFlyingClawedFishEating = HasClaws | CanFly | EatsFish | Endangered
}
```

字符串枚举

在上文中，我们只看到了数字类型的枚举，实际上，枚举类型的值，也可以是字符串类型。

```
export enum EvidenceTypeEnum {
  UNKNOWN = '',
  PASSPORT_VISA = 'passport_visa',
  PASSPORT = 'passport',
  SIGHTED_STUDENT_CARD = 'sighted_tertiary_edu_id',
  SIGHTED_KEYPASS_CARD = 'sighted_keypass_card',
  SIGHTED_PROOF_OF_AGE_CARD = 'sighted_proof_of_age_card'
}
```

这些可以更容易被处理和调试，因为它们提供有意义/可调试的字符串。

你可以使用它们用于简单的字符串比较：

```
// Where `someStringFromBackend` will be '' | 'passport_visa' | 'passport' ... etc.
const value = someStringFromBackend as EvidenceTypeEnum;

// Sample use in code
if (value === EvidenceTypeEnum.PASSPORT) {
  console.log('You provided a passport');
  console.log(value); // `passport`
}
```

常量枚举

```
enum Tristate {
  False,
  True,
  Unknown
}

const lie = Tristate.False;
```

`const lie = Tristate.False` 编译成 JavaScript `let lie = Tristate.False` (是的，编译后与编译前，几乎相同)。这意味着在运行执行时，它将会查找变量 `Tristate` 和 `Tristate.False` 。在此处获得性能提升的一个小技巧是使用常量枚举：

```
const enum Tristate {
```

```
  False,
  True,
  Unknown
}

const lie = Tristate.False;
```

将会被编译成：

```
let lie = 0;
```

编译器将会：

- 内联枚举的任何用法（`0` 而不是 `Tristate.False`）；
- 不会为枚举类型编译成任何 JavaScript（在这个例子中，运行时没有 `Tristate` 变量），因为它使用内联语法。

常量枚举 preserveConstEnums 选项

使用内联语法对性能有明显的提升作用。运行时没有 `Tristate` 变量的事实，是因为编译器帮助你把一些在运行时没有用到的不编译成 JavaScript。然而，你可能想让编译器仍然把枚举类型编译成 JavaScript，用于如上例子中从字符串到数字，或者是从数字到字符串的查找。在这种情景下，你可以使用编译选项 `—preserveConstEnums`，它会编译出 `var Tristate` 的定义，因此你在运行时，手动使用 `Tristate['False']` 和 `Tristate[0]`。并且这不会以任何方式影响内联。

有静态方法的枚举

你可以使用 `enum` + `namespace` 的声明的方式向枚举类型添加静态方法。如下例所示，我们将静态成员 `isBusinessDay` 添加到枚举上：

```
enum Weekday {
  Monday,
  Tuseday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday
}

namespace Weekday {
```

```
  export function isBusinessDay(day: Weekday) {
    switch (day) {
      case Weekday.Saturday:
      case Weekday.Sunday:
        return false;
      default:
        return true;
    }
  }
}

const mon = Weekday.Monday;
const sun = Weekday.Sunday;

console.log(Weekday.isBusinessDay(mon)); // true
console.log(Weekday.isBusinessDay(sun));
```

开放式枚举

TIP

你只有在不使用模块时，开放式的枚举才有意义，你应该使用模块，因此这部分在文章最后。

让我们再一次看看编译成 JavaScript 的枚举是什么样子：

```
var Tristate;
(function(Tristate) {
  Tristate[(Tristate['False'] = 0)] = 'False';
  Tristate[(Tristate['True'] = 1)] = 'True';
  Tristate[(Tristate['Unknown'] = 2)] = 'Unknown';
})(Tristate || (Tristate = {}));
```

我们已经解释了 `Tristate[Tristate['False'] = 0] = 'False'` 部分，现在我们来看看包裹函数 `(function (Tristate) { / code here /})(Tristate || (Tristate = {}))` ，特别是 `(Tristate || (Tristate = {}))` 部分。这捕获了一个局部变量 `TriState` ，它将指向已经定义的 `TriState` 或者是使用一个新的控对象初始化它。

这意味着你可以跨多个文件拆分（和扩展）枚举定义，如下所示，你可以把 `Color` 的定义拆分至两个块中：

```
enum Color {
  Red,
```

```
  Green,
  Blue
}

enum Color {
  DarkRed = 3,
  DarkGreen,
  DarkBlue
}
```

TIP

你应该在枚举的延续块中，初始化第一个成员，以便生成的代码不是先前定义的枚举类型值。TypeScript 将会发出警告，如果你定义初始值（错误信息：`In an enum with multiple declarations, only one declaration can omit an initializer for its first enum element.`）。

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/enums.html*

lib.d.ts

当你安装 `TypeScript` 时，会顺带安装 `lib.d.ts` 等声明文件。此文件包含了 JavaScript 运行时以及 DOM 中存在各种常见的环境声明。

- 它自动包含在 TypeScript 项目的编译上下文中；
- 它能让你快速开始书写经过类型检查的 JavaScript 代码。
  你可以通过指定 `—noLib` 的编译器命令行标志（或者在 `tsconfig.json` 中指定选项 `noLib`: true）从上下文中排除此文件。

使用例子

看如下例子：

```
const foo = 123;
const bar = foo.toString();
```

这段代码的类型检查正常，因为 `lib.d.ts` 为所有 JavaScript 对象定义了 `toString` 方法。

如果你在 `noLib` 选项下，使用相同的代码，这将会出现类型检查错误：

```
const foo = 123;
const bar = foo.toString(); // Error: 属性 toString 不存在类型 number 上
```

现在你已经理解了 `lib.d.ts` 的重要性，至于它的内容是怎么样的，我们接下来解释。

观察 lib.d.ts 的内部

`lib.d.ts` 的内容主要是一些变量声明（如：`window`、`document`、`math`）和一些类似的接口声明（如：`Window`、`Document`、`Math`）。

最简单的方式寻找代码的类型（如：`Math.floor`）是使用 IDE 的 `F12`（跳转到定义）。

让我们来看一个示例变量的声明，如 `window` 被定义为：

```
declare var window: Window;
```

这只是一个简单的 `declare var`，后面跟一个变量名称（`window`）和一个用来类型注解的接口（`Window` 接口），这些变量通常指向一些全局的接口，例如，以下是 `Window` 接口的一小部

分：

```
interface Window extends EventTarget, WindowTimers, WindowSessionStorage,
WindowLocalStorage, WindowConsole, GlobalEventHandlers, IDBEnvironment,
WindowBase64 {
  animationStartTime: number;
  applicationCache: ApplicationCache;
  clientInformation: Navigator;
  closed: boolean;
  crypto: Crypto;
  // so on and so forth...
}
```

你可以在这些接口里看到大量的类型信息，当你不使用 TypeScript 时，你需要将它们保存在你的大脑里。现在你可以使用 `intellisense` 之类东西，从而可以减少对知识的记忆。

使用这些全局变量是有利的。在不更改 `lib.d.ts` 的情景下，它可以让你添加额外的属性。接下来，我们将介绍这些概念。

修改原始类型

在 TypeScript 中，接口是开放式的，这意味着当你想使用不存在的成员时，你仅仅是需要添加它们至 `lib.d.ts` 中的接口声明中，TypeScript 将会自动接收它。注意，你需要在全局模块中做这些修改，以使这些接口与 `lib.d.ts` 相关联。我们推荐你创建一个称为 `globals.d.ts` 的特殊文件。

这里有我们需要添加至 `Window`，`Math`，`Date` 的一些例子：

Window

仅仅是添加至 `Window` 接口：

```
interface Window {
  helloWorld(): void;
}
```

这将允许你以一种类型安全的形式使用它：

```
// Add it at runtime
window.helloWorld = () => console.log('hello world');

```

```
// Call it
window.helloWorld();

// 滥用会导致错误
window.helloWorld('gracius'); // Error: 提供的参数与目标不匹配
```

Math

全局变量 `Math` 在 `lib.d.ts` 中被定义为：

```
/** An intrinsic object that provides basic mathematics functionality and constants. */
declare var Math: Math;
```

即变量 `Math` 是 `Math` 的一个实例，`Math` 接口被定义为

```
interface Math {
  E: number;
  LN10: number;
  // others ...
}
```

当你想在 `Math` 全局变量上添加你需要的属性时，你仅需要把它添加至 `Math` 的全局接口上即可，例如：在 `seedrandom Project` 项目里，它添加了 `seedrandom` 函数至全局的 `Math` 对象上，这可以很容易的被声明：

```
interface Math {
  seedrandom(seed?: string): void;
}
```

你可以像下面一样使用它：

```
Math.seedrandom();

Math.seedrandom('Any string you want');
```

Date

如果你在 `lib.d.ts` 中寻找 `Date` 定义的声明，你将会找到：

```
declare var Date: DateConstructor;
```

接口 `DateConstructor` 与你在上文中看到的 `Math` 和 `Window` 接口一样，因为它涵盖了可以使用的 `Date` 全局变量的成员（如：`Date.now()`）。除此之外，它还包含了可以让你创建 `Date` 实例的构造函数签名（如：`new Date()`）。`DateConstructor` 接口的一部分代码如下所示：

```
interface DateConstructor {
  new (): Date;
  // 一些其他的构造函数签名

  now(): number;

  // 其他成员函数
}
```

在 dayjs 里，它在 `Date` 的全局变量以及 `Date` 实例上同时添加了成员，因此这个库的 TypeScript 定义看起来像如下所示（社区已经定义好了）：

```
// DateJS 公开的静态方法
interface DateConstructor {
  /** Gets a date that is set to the current date. The time is set to the start of the day (00:00 or 12:00 AM) */
  today(): Date;
  // ... so on and so forth
}

// DateJS 公开的实例方法
interface Date {
  /** Adds the specified number of milliseconds to this instance. */
  addMilliseconds(milliseconds: number): Date;
  // ... so on and so forth
}
```

这允许你在类型安全的情况下做：

```
const today = Date.today();
const todayAfter1second = today.addMilliseconds(1000);
```

string

如果你在 `lib.d.ts` 里寻找 `string`，你将会找到与 `Date` 相类似的内容（全局变量

`String` , `StringConstructor` 接口, `String` 接口）。但是值得注意的是, `String` 接口也会影响字符串字面量，如下所示:

```
interface String {
  endsWith(suffix: string): boolean;
}

String.prototype.endsWith = function(suffix: string): boolean {
  const str: string = this;
  return str && str.indexOf(suffix, str.length - suffix.length) !== -1;
};

console.log('foo bar'.endsWith('bas')); // false
console.log('foo bas'.endsWith('bas')); // true
```

终极 string

基于可维护性，我们推荐创建一个 `global.d.ts` 文件。然而，如果你愿意，你可以通过使用 `declare global { / global namespace / }` ，从文件模块中进入全局命名空间:

```
// 确保是模块
export {};

declare global {
  interface String {
    endsWith(suffix: string): boolean;
  }
}

String.prototype.endsWith = function(suffix: string): boolean {
  const str: string = this;
  return str && str.indexOf(suffix, str.length - suffix.length) !== -1;
};

console.log('foo bar'.endsWith('bas')); // false
console.log('foo bas'.endsWith('bas')); // true
```

使用你自己定义的 lib.d.ts

如上文说提及，使用 `—noLib` 编译选项会导致 TypeScript 排除自动包含的 `lib.d.ts` 文

件。为什么这个功能是有效的，我例举了一些常见原因：

- 运行的 JavaScript 环境与基于标准浏览器运行时环境有很大不同；
- 您希望在代码里严格的控制全局变量，例如：`lib.d.ts` 定义了 `item` 作为全局变量，你并不希望它泄漏到你的代码里。
  一旦你排除了默认的 `lib.d.ts` 文件，你可以在编译上下文中包含一个类似命名的文件，TypeScript 将选择它进行类型检查。

TIP

小心使用 `—noLib` 选项，一旦你使用使用了它，当你把你的项目分享给其他人时，它们也将被迫使用 `—noLib` 选项，更糟糕的是，如果将这些代码放入你的项目中，你可能需要将它放入你的基于 `lib` 代码中。

编译目标对 lib.d.ts 的影响

设置编译目标为 `es6` 时，能导致 `lib.d.ts` 包含更多的像 Promise 的现代（es6）内容的环境声明。编译器目标的这种神奇作用，改变了代码的环境，这对某些人来说是理想的，但是这对另外一些人来说造成了困扰，因为它将编译出的代码与环境混为一谈。

当你想对环境进行更细粒的控制时，你应该使用我们接下来将要讨论的 `—lib` 选项。

—lib 选项

一些时候，你想要解耦编译目标（生成的 JavaScript 版本）和环境库支持之间的关系。例如对于 Promise，你的编译目标是 `—target es5`，但是你仍然想使用它，这个时候，你可以使用 `lib` 对它进行控制。

TIP

使用 `—lib` 选项可以将任何 `lib` 与 `—target` 解偶。

你可以通过命令行或者在 `tsconfig.json` 中提供此选项（推荐）：

命令行

```
tsc --target es5 --lib dom,es6
```

config.json

```
"compilerOptions": {
```

```
    "lib": ["dom", "es6"]
}
```

`lib` 分类如下:

- JavaScript 功能
  - es5
  - es6
  - es2015
  - es7
  - es2016
  - es2017
  - esnext
- 运行环境
  - dom
  - dom.iterable
  - webworker
  - scripthost
- ESNext 功能选项
  - es2015.core
  - es2015.collection
  - es2015.generator
  - es2015.iterable
  - es2015.promise
  - es2015.proxy
  - es2015.reflect
  - es2015.symbol
  - es2015.symbol.wellknown
  - es2016.array.include
  - es2017.object
  - es2017.sharedmemory
  - esnext.asynciterable

NOTE

`–lib` 选项提供非常精细的控制，因此你最有可能从运行环境与 JavaScript 功能类别中分别选择一项，如果你没有指定 `–lib`，则会导入默认库:

- `–target` 选项为 es5 时，会导入 es5, dom, scripthost。
- `–target` 选项为 es6 时，会导入 es6, dom, dom.iterable, scripthost。

我个人的推荐:

```
"compilerOptions": {
  "target": "es5",
  "lib": ["es6", "dom"]
}
```

包括使用 Symbol 的 ES5 使用例子:

```
"compilerOptions": {
  "target": "es5",
  "lib": ["es5", "dom", "scripthost", "es2015.symbol"]
}
```

在旧的 JavaScript 引擎时使用 Polyfill

> 关于此主题的一个视频

要使用一些新功能如 `Map`、`Set`、`Promise`（随着时间推移会变化），你可以使用现代的 `lib` 选项，并且需要安装 `core-js`：

```
npm install core-js --save-dev
```

接着，在你的项目里导入它:

```
import 'core-js';
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/lib.html*

函数

函数类型在 TypeScript 类型系统中扮演着非常重要的角色，它们是可组合系统的核心构建块。

参数注解

你可以注解函数参数，就像你可以注解其他变量一样：

```
// variable annotation
let sampleVariable: { bar: number };

// function parameter annotation
function foo(sampleParameter: { bar: number }) {}
```

这里我们使用了内联类型注解，除此之外，你还可以使用接口等其他方式。

返回类型注解

你可以在函数参数列表之后使用与变量相同的样式来注解返回类型，如例子中 `: Foo`：

```
interface Foo {
  foo: string;
}

// Return type annotated as `: Foo`
function foo(sample: Foo): Foo {
  return sample;
}
```

我们在这里使用了一个 `interface`，但你可以自由地使用其他注解方式，例如内联注解。

通常，你不需要注解函数的返回类型，因为它可以由编译器推断：

```
interface Foo {
  foo: string;
}

function foo(sample: Foo) {
  return sample; // inferred return type 'Foo'
}
```

但是，添加这些注解以帮助解决错误提示通常是一个好主意，例如：

```
function foo() {
  return { fou: 'John Doe' }; // You might not find this misspelling of `foo` till it's too late
}

sendAsJSON(foo());
```

如果你不打算从函数返回任何内容，则可以将其标注为：`void`。你通常可以删除 `void`，TypeScript 能推导出来：

可选参数

你可以将参数标记为可选：

```
function foo(bar: number, bas?: string): void {
  // ..
}

foo(123);
foo(123, 'hello');
```

或者，当调用者没有提供该参数时，你可以提供一个默认值（在参数声明后使用 `= someValue`）：

```
function foo(bar: number, bas: string = 'hello') {
  console.log(bar, bas);
}

foo(123); // 123, hello
foo(123, 'world'); // 123, world
```

重载

TypeScript 允许你声明函数重载。这对于文档 + 类型安全来说很实用。请思考以下代码：

```
function padding(a: number, b?: number, c?: number, d?: any) {
  if (b === undefined && c === undefined && d === undefined) {
    b = c = d = a;
  } else if (c === undefined && d === undefined) {
```

```
    c = a;
    d = b;
  }
  return {
    top: a,
    right: b,
    bottom: c,
    left: d
  };
}
```

如果仔细查看代码，就会发现 a，b，c，d 的值会根据传入的参数数量而变化。此函数也只需要 1 个，2 个或 4 个参数。可以使用函数重载来强制和记录这些约束。你只需多次声明函数头。最后一个函数头是在函数体内实际处于活动状态但不可用于外部。

如下所示：

```
// 重载
function padding(all: number);
function padding(topAndBottom: number, leftAndRight: number);
function padding(top: number, right: number, bottom: number, left: number);
// Actual implementation that is a true representation of all the cases the function body needs to handle
function padding(a: number, b?: number, c?: number, d?: number) {
  if (b === undefined && c === undefined && d === undefined) {
    b = c = d = a;
  } else if (c === undefined && d === undefined) {
    c = a;
    d = b;
  }
  return {
    top: a,
    right: b,
    bottom: c,
    left: d
  };
}
```

这里前三个函数头可有效调用 `padding`：

```
padding(1); // Okay: all
padding(1, 1); // Okay: topAndBottom, leftAndRight
```

```
padding(1, 1, 1, 1); // Okay: top, right, bottom, left

padding(1, 1, 1); // Error: Not a part of the available overloads
```

当然，最终声明（从函数内部看到的真正声明）与所有重载兼容是很重要的。这是因为这是函数体需要考虑的函数调用的真实性质。

TIP

TypeScript 中的函数重载没有任何运行时开销。它只允许你记录希望调用函数的方式，并且编译器会检查其余代码。

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/functions.html*

可调用的

你可以使用类型别名或者接口来表示一个可被调用的类型注解：

```
interface ReturnString {
  (): string;
}
```

它可以表示一个返回值为 `string` 的函数：

```
declare const foo: ReturnString;

const bar = foo(); // bar 被推断为一个字符串。
```

一个实际的例子

当然，像这样一个可被调用的类型注解，你也可以根据实际来传递任何参数、可选参数以及 rest 参数，这有一个稍微复杂的例子：

```
interface Complex {
  (foo: string, bar?: number, ...others: boolean[]): number;
}
```

一个接口可提供多种调用签名，用以特殊的函数重载：

```
interface Overloaded {
  (foo: string): string;
  (foo: number): number;
}

// 实现接口的一个例子：
function stringOrNumber(foo: number): number;
function stringOrNumber(foo: string): string;
function stringOrNumber(foo: any): any {
  if (typeof foo === 'number') {
    return foo * foo;
  } else if (typeof foo === 'string') {
    return `hello ${foo}`;
  }
```

```
}

const overloaded: Overloaded = stringOrNumber;

// 使用
const str = overloaded(''); // str 被推断为 'string'
const num = overloaded(123); // num 被推断为 'number'
```

这也可以用于内联注解中：

```
let overloaded: {
  (foo: string): string;
  (foo: number): number;
};
```

箭头函数

为了使指定可调用的类型签名更容易，TypeScript 也允许你使用简单的箭头函数类型注解。例如，在一个以 number 类型为参数，以 string 类型为返回值的函数中，你可以这么写：

```
const simple: (foo: number) => string = foo => foo.toString();
```

TIP

它仅仅只能做为简单的箭头函数，你无法使用重载。如果想使用它，你必须使用完整的 `{ (someArgs): someReturn }` 的语法

可实例化

可实例化仅仅是可调用的一种特殊情况，它使用 `new` 做为前缀。它意味着你需用使用 `new` 关键字去调用它：

```
interface CallMeWithNewToGetString {
  new (): string;
}

// 使用
declare const Foo: CallMeWithNewToGetString;
const bar = new Foo(); // bar 被推断为 string 类型
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/callable.html*

类型断言

TypeScript 允许你覆盖它的推断，并且能以你任何你想要的方式分析它，这种机制被称为「类型断言」。TypeScript 类型断言用来告诉编译器你比它更了解这个类型，并且它不应该再发出错误。

类型断言的一个常见用例是当你从 JavaScript 迁移到 TypeScript 时：

```
const foo = {};
foo.bar = 123; // Error: 'bar' 属性不存在于 ‘{}’
foo.bas = 'hello'; // Error: 'bas' 属性不存在于 '{}'
```

这里的代码发出了错误警告，因为 `foo` 的类型推断为 `{}`，即是具有零属性的对象。因此，你不能在它的属性上添加 `bar` 或 `bas`，你可以通过类型断言来避免此问题：

```
interface Foo {
  bar: number;
  bas: string;
}

const foo = {} as Foo;
foo.bar = 123;
foo.bas = 'hello';
```

as foo 与 <foo>

最初的断言语法如下所示：

```
let foo: any;
let bar = <string>foo; // 现在 bar 的类型是 'string'
```

然而，当你在 JSX 中使用 `<foo>` 的断言语法时，这会与 JSX 的语法存在歧义：

```
let foo = <string>bar;</string>;
```

因此，为了一致性，我们建议你使用 `as foo` 的语法来为类型断言。

类型断言与类型转换

它之所以不被称为「类型转换」，是因为转换通常意味着某种运行时的支持。但是，类型断言纯粹是一

个编译时语法，同时，它也是一种为编译器提供关于如何分析代码的方法。

类型断言被认为是有害的

在很多情景下，断言能让你更容易的从遗留项目中迁移（甚至将其他代码粘贴复制到你的项目中），然而，你应该小心谨慎的使用断言。让我们用最初的代码做为示例，如果你没有按约定添加属性，TypeScript 编译器并不会对此发出错误警告：

```
interface Foo {
  bar: number;
  bas: string;
}

const foo = {} as Foo;

// ahhh, 忘记了什么？
```

另外一个常见的想法是使用类型断言来提供代码的提示：

```
interface Foo {
  bar: number;
  bas: string;
}

const foo = <Foo>{
  // 编译器将会提供关于 Foo 属性的代码提示
  // 但是开发人员也很容易忘记添加所有的属性
  // 同样，如果 Foo 被重构，这段代码也可能被破坏（例如，一个新的属性被添加）。
};
```

这也会存在一个同样的问题，如果你忘记了某个属性，编译器同样也不会发出错误警告。使用一种更好的方式：

```
interface Foo {
  bar: number;
  bas: string;
}

const foo: Foo = {
  // 编译器将会提供 Foo 属性的代码提示
};
```

在某些情景下，你可能需要创建一个临时的变量，但至少，你将不会使用一个承若（可能是假的），而是依靠类型推断来检查你的代码。

双重断言

类型断言，尽管我们已经证明了它并不是那么安全，但是它也不是完全不安全的。如下一个非常实用的例子所示，当使用者了解传入参数更具体的类型时，类型断言能按预期工作：

```
function handler(event: Event) {
  const mouseEvent = event as MouseEvent;
}
```

然而，如下例子中的代码将会报错，尽管使用者已经使用了类型断言：

```
function handler(event: Event) {
  const element = event as HTMLElement; // Error: 'Event' 和 'HTMLElement' 中的任何一个都不能赋值给另外一个
}
```

如果你仍然想使用那个类型，你可以使用双重断言。首先断言成兼容所有类型的 `any`，编译器将不会报错：

```
function handler(event: Event) {
  const element = (event as any) as HTMLElement; // ok
}
```

TypeScript 是怎么确定单个断言是否足够

当 `S` 类型是 `T` 类型的子集，或者 `T` 类型是 `S` 类型的子集时，`S` 能被成功断言成 `S`。这是为了在进行类型断言时提供额外的安全性，完全毫无根据的断言时危险的，如果你想这么做，你可以使用 `any`。

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/typeAssertion.html*

Freshness

为了能让检查对象字面量类型更容易，TypeScript 提供 「Freshness」 的概念（它也被称为更严格的对象字面量检查）用来确保对象字面量在结构上类型兼容。

结构类型非常方便。考虑如下例子代码，它可以让你非常便利的从 JavaScript 迁移至 TypeScript，并且会提供类型安全：

```
function logName(something: { name: string }) {
  console.log(something.name);
}

const person = { name: 'matt', job: 'being awesome' };
const animal = { name: 'cow', diet: 'vegan, but has milk of own specie' };
const randow = { note: `I don't have a name property` };

logName(person); // ok
logName(animal); // ok
logName(randow); // Error: 没有 `name` 属性
```

但是，结构类型有一个缺点，它能误导你认为某些东西接收的数据比它实际的多。如下例，TypeScript 发出错误警告：

```
function logName(something: { name: string }) {
  console.log(something.name);
}

logName({ name: 'matt' }); // ok
logName({ name: 'matt', job: 'being awesome' }); // Error: 对象字面量只能指定已知属性，`job` 属性在这里并不存在。
```

WARNING

请注意，这种错误提示，只会发生在对象字面量上。

如果没有这种错误提示，我们可能会去寻找函数的调用 `logName({ name: 'matt', job: 'being awesome' })`，继而会认为 `logName` 可能会使用 `job` 属性做一些事情，然而实际上 `logName` 并没有使用它。

另外一个使用比较多的场景是与具有可选成员的接口一起使用，如果没有这样的对象字面量检查，当你输入错误单词的时候，并不会发出错误警告：

```
function logIfHasName(something: { name?: string }) {
  if (something.name) {
    console.log(something.name);
  }
}

const person = { name: 'matt', job: 'being awesome' };
const animal = { name: 'cow', diet: 'vegan, but has milk of own species' };

logIfHasName(person); // okay
logIfHasName(animal); // okay

logIfHasName({ neme: 'I just misspelled name to neme' }); // Error: 对象字面量只能指定已知属性，`neme` 属性不存在。
```

之所以只对对象字面量进行类型检查，因为在这种情况下，那些实际上并没有被使用到的属性有可能会拼写错误或者会被误用。

允许额外的属性

一个类型能够包含索引签名，以明确表明可以使用额外的属性：

```
let x: { foo: number, [x: string]: any };

x = { foo: 1, baz: 2 }; // ok, 'baz' 属性匹配于索引签名
```

用例：React State

Facebook ReactJS 为对象的 Freshness 提供了一个很好的用例，通常在组件中，你只使用少量属性，而不是传入所有，来调用 `setState`：

```
// 假设
interface State {
  foo: string;
  bar: string;
}

// 你可能想做：
this.setState({ foo: 'Hello' }); // Error: 没有属性 'bar'

```

```
// 因为 state 包含 'foo' 与 'bar'，TypeScript 会强制你这么做：
this.setState({ foo: 'Hello', bar: this.state.bar });
```

如果你想使用 Freshness，你可能需要将所有成员标记为可选，这仍然会捕捉到拼写错误：

```
// 假设
interface State {
  foo?: string;
  bar?: string;
}

// 你可能想做
this.setState({ foo: 'Hello' }); // Yay works fine!

// 由于 Freshness，你也可以防止错别字
this.setState({ foos: 'Hello' }}; // Error: 对象只能指定已知属性

// 仍然会有类型检查
this.setState({ foo: 123 }}; // Error: 无法将 number 类型赋值给 string 类型
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/freshness.html*

类型保护

类型保护允许你使用更小范围下的对象类型。

typeof

TypeScript 熟知 JavaScript 中 `instanceof` 和 `typeof` 运算符的用法。如果你在一个条件块中使用这些，TypeScript 将会推导出在条件块中的的变量类型。如下例所示，TypeScript 将会辨别 `string` 上是否存在特定的函数，以及是否发生了拼写错误：

```
function doSome(x: number | string) {
  if (typeof x === 'string') {
    // 在这个块中，TypeScript 知道 `x` 的类型必须是 `string`
    console.log(x.subtr(1)); // Error: 'subtr' 方法并没有存在于 `string` 上
    console.log(x.substr(1)); // ok
  }

  x.substr(1); // Error: 无法保证 `x` 是 `string` 类型
}
```

instanceof

这有一个关于 `class` 和 `instanceof` 的例子：

```
class Foo {
  foo = 123;
  common = '123';
}

class Bar {
  bar = 123;
  common = '123';
}

function doStuff(arg: Foo | Bar) {
  if (arg instanceof Foo) {
    console.log(arg.foo); // ok
    console.log(arg.bar); // Error
  }
```

```
  if (arg instanceof Bar) {
    console.log(arg.foo); // Error
    console.log(arg.bar); // ok
  }
}

doStuff(new Foo());
doStuff(new Bar());
```

TypeScript 甚至能够理解 `else`。当你使用 `if` 来缩小类型时，TypeScript 知道在其他块中的类型并不是 `if` 中的类型：

```
class Foo {
  foo = 123;
}

class Bar {
  bar = 123;
}

function doStuff(arg: Foo | Bar) {
  if (arg instanceof Foo) {
    console.log(arg.foo); // ok
    console.log(arg.bar); // Error
  } else {
    // 这个块中，一定是 'Bar'
    console.log(arg.foo); // Error
    console.log(arg.bar); // ok
  }
}

doStuff(new Foo());
doStuff(new Bar());
```

in

`in` 操作符可以安全的检查一个对象上是否存在一个属性，它通常也被做为类型保护使用：

```
interface A {
  x: number;
}
```

```

interface B {
  y: string;
}

function doStuff(q: A | B) {
  if ('x' in q) {
    // q: A
  } else {
    // q: B
  }
}
```

字面量类型保护

当你在联合类型使里使用字面量类型时，你可以检查它们是否有区别：

```
type Foo = {
  kind: 'foo'; // 字面量类型
  foo: number;
};

type Bar = {
  kind: 'bar'; // 字面量类型
  bar: number;
};

function doStuff(arg: Foo | Bar) {
  if (arg.kind === 'foo') {
    console.log(arg.foo); // ok
    console.log(arg.bar); // Error
  } else {
    // 一定是 Bar
    console.log(arg.foo); // Error
    console.log(arg.bar); // ok
  }
}
```

使用定义的类型保护

JavaScript 并没有内置非常丰富的、运行时的自我检查机制。当你在使用普通的 JavaScript 对象时（使用结构类型，更有益处），你甚至无法访问 `instacneof` 和 `typeof` 。在这种情景下，你可以创建用户自定义的类型保护函数，这仅仅是一个返回值为类似于 `someArgumentName is SomeType` 的函数，如下：

```
// 仅仅是一个 interface
interface Foo {
  foo: number;
  common: string;
}

interface Bar {
  bar: number;
  common: string;
}

// 用户自己定义的类型保护！
function isFoo(arg: Foo | Bar): arg is Foo {
  return (arg as Foo).foo !== undefined;
}

// 用户自己定义的类型保护使用用例：
function doStuff(arg: Foo | Bar) {
  if (isFoo(arg)) {
    console.log(arg.foo); // ok
    console.log(arg.bar); // Error
  } else {
    console.log(arg.foo); // Error
    console.log(arg.bar); // ok
  }
}

doStuff({ foo: 123, common: '123' });
doStuff({ bar: 123, common: '123' });
```

> 原文：*https://jkchao.github.io/typescript-book-chinese/typings/typeGuard.html*

字面量类型

字面量是 JavaScript 本身提供的一个准确变量。

字符串字面量

你可以使用一个字符串字面量做为一个类型：

```
let foo: 'Hello';
```

在这里，我们创建了一个被称为 `foo` 变量，它仅接收一个字面量值为 `Hello` 的变量：

```
let foo: 'Hello';
foo = 'Bar'; // Error: 'bar' 不能赋值给类型 'Hello'
```

它们本身并不是很实用，但是可以在一个联合类型中组合创建一个强大的（实用的）抽象：

```
type CardinalDirection = 'North' | 'East' | 'South' | 'West';

function move(distance: number, direction: CardinalDirection) {
  // ...
}

move(1, 'North'); // ok
move(1, 'Nurth'); // Error
```

其他字面量类型

TypeScript 同样也提供 `boolean` 和 `number` 的字面量类型：

```
type OneToFive = 1 | 2 | 3 | 4 | 5;
type Bools = true | false;
```

推断

通常，你会得到一个类似于 `Type string is not assignable to type 'foo'` 的错误，如下：

```
function iTakeFoo(foo: 'foo') {}
```

```
const test = {
  someProp: 'foo'
};

iTakeFoo(test.someProp); // Error: Argument of type string is not assignable to parameter of type 'foo'
```

这是由于 `test` 被推断为 `{ someProp: string }`，我们可以采用一个简单的类型断言来告诉 TypeScript 你想推断的字面量：

```
function iTakeFoo(foo: 'foo') {}

const test = {
  someProp: 'foo' as 'foo'
};

iTakeFoo(test.someProp); // ok
```

或者使用类型注解的方式，来帮助 TypeScript 推断正确的类型：

```
function iTakeFoo(foo: 'foo') {}

type Test = {
  someProp: 'foo';
};

const test: Test = {
  // 推断 `someProp` 永远是 'foo'
  someProp: 'foo'
};

iTakeFoo(test.someProp); // ok
```

使用用例

TypeScript 枚举类型是基于数字的，你可以使用带字符串字面量的联合类型，来模拟一个基于字符串的枚举类型，就好像上文中提出的 `CardinalDirection` 。你甚至可以使用下面的函数来生成 `key: value` 的结构：

```
// 用于创建字符串列表映射至 `K: V` 的函数
function strEnum<T extends string>(o: Array<T>): { [K in T]: K } {
```

```
  return o.reduce((res, key) => {
    res[key] = key;
    return res;
  }, Object.create(null));
}
```

然后，你就可以使用 `keyof`、`typeof` 来生成字符串的联合类型。下面是一个完全的例子：

```
// 用于创建字符串列表映射至 `K: V` 的函数
function strEnum<T extends string>(o: Array<T>): { [K in T]: K } {
  return o.reduce((res, key) => {
    res[key] = key;
    return res;
  }, Object.create(null));
}

// 创建 K: V
const Direction = strEnum(['North', 'South', 'East', 'West']);

// 创建一个类型
type Direction = keyof typeof Direction;

// 简单的使用
let sample: Direction;

sample = Direction.North; // Okay
sample = 'North'; // Okay
sample = 'AnythingElse'; // ERROR!
```

辨析联合类型

我们将会在此书的稍后章节讲解它。

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/literals.html*

readonly

TypeScript 类型系统允许你在一个接口里使用 `readonly` 来标记属性。它能让你以一种更安全的方式工作（不可预期的改变是很糟糕的）：

```
function foo(config: { readonly bar: number, readonly bas: number }) {
  // ..
}

const config = { bar: 123, bas: 123 };
foo(config);

// 现在你能够确保 'config' 不能够被改变了
```

当然，你也可以在 `interface` 和 `type` 里使用 `readonly`：

```
type Foo = {
  readonly bar: number;
  readonly bas: number;
};

// 初始化
const foo: Foo = { bar: 123, bas: 456 };

// 不能被改变
foo.bar = 456; // Error: foo.bar 为仅读属性
```

你也能指定一个类的属性为只读，然后在声明时或者构造函数中初始化它们，如下所示：

```
class Foo {
  readonly bar = 1; // OK
  readonly baz: string;
  constructor() {
    this.baz = 'hello'; // OK
  }
}
```

Readonly

这有一个 `Readonly` 的映射类型，它接收一个泛型 `T`，用来把它的所有属性标记为只读类型：

```
type Foo = {
  bar: number;
  bas: number;
};

type FooReadonly = Readonly<Foo>;

const foo: Foo = { bar: 123, bas: 456 };
const fooReadonly: FooReadonly = { bar: 123, bas: 456 };

foo.bar = 456; // ok
fooReadonly.bar = 456; // Error: bar 属性只读
```

其他的使用用例

ReactJS

`ReactJS` 是一个喜欢用不变数据的库，你可以标记你的 `Props` 和 `State` 为不可变数据：

```
interface Props {
  readonly foo: number;
}

interface State {
  readonly bar: number;
}

export class Something extends React.Component<Props, State> {
  someMethod() {
    // 你可以放心，没有人会像下面这么做
    this.props.foo = 123; // Error: props 是不可变的
    this.state.baz = 456; // Error: 你应该使用 this.setState()
  }
}
```

然而，你并没有必要这么做， `React` 的声明文件已经标记这些为 `readonly` （通过传入泛型参数至一个内部包装，来把每个属性标记为 `readonly` ，如上例子所示），

```
export class Something extends React.Component<{ foo: number }, { baz: number
```

```
}> {
  someMethod() {
    this.props.foo = 123; // Error: props 是不可变的
    this.state.baz = 456; // Error: 你应该使用 this.setState()
  }
}
```

绝对的不可变

你甚至可以把索引签名标记为只读：

```
interface Foo {
  readonly [x: number]: number;
}

// 使用

const foo: Foo = { 0: 123, 2: 345 };
console.log(foo[0]); // ok（读取）
foo[0] = 456; // Error: 属性只读
```

如果你想以不变的方式使用原生 JavaScript 数组，可以使用 TypeScript 提供的 `ReadonlyArray<T>` 接口：

```
let foo: ReadonlyArray<number> = [1, 2, 3];
console.log(foo[0]); // ok
foo.push(4); // Error: ReadonlyArray 上不存在 `push`，因为他会改变数组
foo = foo.concat(4); // ok, 创建了一个复制
```

自动推断

在一些情况下，编译器能把一些特定的属性推断为 `readonly`，例如在一个 `class` 中，如果你有一个只含有 `getter` 但是没有 `setter` 的属性，他能被推断为只读：

```
class Person {
  firstName: string = 'John';
  lastName: string = 'Doe';

  get fullName() {
    return this.firstName + this.lastName;
  }
```

```
}

const person = new Person();

console.log(person.fullName); // John Doe
person.fullName = 'Dear Reader'; // Error, fullName 只读
```

与 const 的不同

`const`

- 用于变量；
- 变量不能重新赋值给其他任何事物。
  `readonly`
- 用于属性；
- 用于别名，可以修改属性；
  简单的例子 1：

```
const foo = 123; // 变量
let bar: {
  readonly bar: number; // 属性
};
```

简单的例子 2：

```
const foo: {
  readonly bar: number;
} = {
  bar: 123
};

function iMutateFoo(foo: { bar: number }) {
  foo.bar = 456;
}

iMutateFoo(foo);
console.log(foo.bar); // 456
```

`readonly` 能确保“我”不能修改属性，但是当你把这个属性交给其他并没有这种保证的使用者（允

许出于类型兼容性的原因），他们能改变它。当然，如果 `iMutateFoo` 明确的表示，他们的参数不可修改，那么编译器会发出错误警告：

```
interface Foo {
  readonly bar: number;
}

let foo: Foo = {
  bar: 123
};

function iTakeFoo(foo: Foo) {
  foo.bar = 456; // Error: bar 属性只读
}

iTakeFoo(foo);
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/readonly.html*

泛型

设计泛型的关键目的是在成员之间提供有意义的约束，这些成员可以是：

- 类的实例成员
- 类的方法
- 函数参数
- 函数返回值

动机和示例

考虑如下简单的 `Queue` （先进先出）数据结构实现，一个在 `TypeScript` 和 `JavaScript` 中的简单实现：

```
class Queue {
  private data = [];
  push = item => this.data.push(item);
  pop = () => this.data.shift;
}
```

在上述代码中存在一个问题，它允许你推入任何类型至队列中，推出的时候也是任意类型，如下所示，但一个人推入一个 `string` 类型至队列中，但是使用者可能会认为队列里只有 `number` 类型：

```
class Queue {
  private data = [];
  push = item => this.data.push(item);
  pop = () => this.data.shift;
}

const queue = new Queue();

queue.push(0);
queue.push('1'); // Oops, 一个错误

// 一个使用者，走入了误区
console.log(queue.pop().toPrecision(1));
console.log(queue.pop().toPrecision(1)); // RUNTIME ERROR
```

一个解决的办法（事实上，这也是不支持泛型类型的唯一解决办法）是为这些约束创建特殊类，如快速创建数字类型的队列：

```
class QueueNumber {
  private data = [];
  push = (item: number) => this.data.push(item);
  pop = (): number => this.data.shift;
}

const queue = new QueueNumber();

queue.push(0);
queue.push('1'); // Error: 不能推入一个 `string` 类型，只能是 `number` 类型

// 如果该错误得到修复，其他将不会出现问题
```

当然，快速意为着痛苦的。例如但你想创建一个字符串的队列时，你将不得不再次修改相当大的代码。我们真正想要的一种方式是无论什么类型被推入队列，被推出的类型都与推入类型一样。当你使用泛型时，这会很容易：

```
// 创建一个泛型类
class Queue<T> {
  private data = [];
  push = (item: T) => this.data.push(item);
  pop = (): T => this.data.shift();
}

// 简单的使用
const queue = new Queue<number>();
queue.push(0);
queue.push('1'); // Error：不能推入一个 `string`，只有 number 类型被允许
```

另外一个我们见过的例子：一个 `reverse` 函数，现在在这个函数里提供了函数参数与函数返回值的约束：

```
function reverse<T>(items: T[]): T[] {
  const toreturn = [];
  for (let i = items.length - 1; i >= 0; i--) {
    toreturn.push(items[i]);
  }
  return toreturn;
}

const sample = [1, 2, 3];
```

```
const reversed = reverse(sample);

reversed[0] = '1'; // Error
reversed = ['1', '2']; // Error

reversed[0] = 1; // ok
reversed = [1, 2]; // ok
```

在此章节中，你已经了解在类和函数上使用泛型的例子。一个值得补充一点的是，你可以为创建的成员函数添加泛型：

```
class Utility {
  reverse<T>(items: T[]): T[] {
    const toreturn = [];
    for (let i = items.length; i >= 0; i--) {
      toreturn.push(items[i]);
    }
    return toreturn;
  }
}
```

TIP

你可以随意调用泛型参数，当你使用简单的泛型时，泛型常用 `T`、`U`、`V` 表示。如果在你的参数里，不止拥有一个泛型，你应该使用一个更语义化名称，如 `TKey` 和 `TValue` （通常情况下，以 `T` 做为泛型前缀也在如 C++ 的其他语言里做为模版。）

误用的泛型

我见过开发者使用泛型仅仅是为了它的 hack。当你使用它时，你应该问问自己：你想用它来提供什么样的约束。如果你不能很好的回答它，你可能会误用泛型，如：

```
declare function foo<T>(arg: T): void;
```

在这里，泛型完全没有必要使用，因为它仅用于单个参数的位置，使用如下方式可能更好：

```
declare function foo(arg: any): void;
```

设计模式：方便通用

考虑如下函数：

```
declare function parse<T>(name: string): T;
```

在这种情况下，泛型 `T` 只在一个地方被使用了，它并没有在成员之间提供约束 `T` 。这相当于一个如下的类型断言：

```
declare function parse(name: string): any;

const something = parse('something') as TypeOfSomething;
```

仅使用一次的泛型并不比一个类型断言来的安全。它们都给你使用 API 提供了便利。

另一个明显的例子是，一个用于加载 json 返回值函数，它返回你任何传入类型的 `Promise` ：

```
const getJSON = <T>(config: { url: string; headers?: { [key: string]: string } }): Promise<T> => {
  const fetchConfig = {
    method: 'GET',
    Accept: 'application/json',
    'Content-Type': 'application/json',
    ...(config.headers || {})
  };
  return fetch(config.url, fetchConfig).then<T>(response => response.json());
};
```

请注意，你仍然需要明显的注解任何你需要的类型，但是 `getJSON<T>` 的签名 `config => Promise<T>` 能够减少你一些关键的步骤（你不需要注解 `loadUsers` 的返回类型，因为它能够被推出来）：

```
type LoadUserResponse = {
  user: {
    name: string;
    email: string;
  }[];
};

function loaderUser() {
  return getJSON<LoadUserResponse>({ url: 'https://example.com/users' });
}
```

与此类似：使用 `Promise<T>` 做为一个函数的返回值比一些如：`Promise<any>` 的备选方案要好很多。

配合 axios 使用

通常情况下，我们会把后端返回数据格式单独放入一个 interface 里：

```
// 请求接口数据
export interface ResponseData<T = any> {
  /**
   * 状态码
   * @type { number }
   */
  code: number;

  /**
   * 数据
   * @type { T }
   */
  result: T;

  /**
   * 消息
   * @type { string }
   */
  message: string;
}
```

当我们把 API 单独抽离成单个模块时：

```
// 在 axios.ts 文件中对 axios 进行了处理，例如添加通用配置、拦截器等
import Ax from './axios';

import { ResponseData } from './interface.ts';

export function getUser<T>() {
  return Ax.get<ResponseData<T>>('/somepath')
    .then(res => res.data)
    .catch(err => console.error(err));
}
```

接着我们写入返回的数据类型 `User` ，这可以让 TypeScript 顺利推断出我们想要的类型：

```
interface User {
  name: string;
  age: number;
}

async function test() {
  // user 被推断出为
  // {
  //  code: number,
  //  result: { name: string, age: number },
  //  message: string
  // }
  const user = await getUser<User>();
}
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/generices.html*

类型推断

TypeScript 能根据一些简单的规则推断（检查）变量的类型，你可以通过实践，很快的了解它们。

定义变量

变量的类型，由定义推断：

```
let foo = 123; // foo 是 'number'
let bar = 'hello'; // bar 是 'string'

foo = bar; // Error: 不能将 'string' 赋值给 `number`
```

这是一个从右向左流动类型的示例。

函数返回类型

返回类型能被 `return` 语句推断，如下所示，推断函数返回为一个数字：

```
function add(a: number, b: number) {
  return a + b;
}
```

这是一个从底部流出类型的例子。

赋值

函数参数类型/返回值也能通过赋值来推断。如下所示，`foo` 的类型是 `Adder`，他能让 `foo` 的参数 `a`、`b` 是 `number` 类型。

```
type Adder = (a: number, b: number) => number;
let foo: Adder = (a, b) => a + b;
```

这个事实可以用下面的代码来证明，TypeScript 会发出正如你期望发出的错误警告：

```
type Adder = (a: number, b: number) => number;
let foo: Adder = (a, b) => {
  a = 'hello'; // Error：不能把 'string' 类型赋值给 'number' 类型
```

```
  return a + b;
};
```

这是一个从左向右流动类型的示例。

如果你创建一个函数，并且函数参数为一个回调函数，相同的赋值规则也适用于它。从 `argument` 至 `parameter` 只是变量赋值的另一种形式。

```
type Adder = (a: number, b: number) => number;
function iTakeAnAdder(adder: Adder) {
  return adder(1, 2);
}

iTakeAnAdder((a, b) => {
  a = 'hello'; // Error: 不能把 'string' 类型赋值给 'number' 类型
  return a + b;
});
```

结构化

这些简单的规则也适用于结构化的存在（对象字面量），例如在下面这种情况下 `foo` 的类型被推断为 `{ a: number, b: number }`：

```
const foo = {
  a: 123,
  b: 456
};

foo.a = 'hello'; // Error：不能把 'string' 类型赋值给 'number' 类型
```

数组也一样：

```
const bar = [1, 2, 3];
bar[0] = 'hello'; // Error：不能把 'string' 类型赋值给 'number' 类型
```

解构

这些也适用于解构中：

```
const foo = {
```

```
  a: 123,
  b: 456
};
let { a } = foo;

a = 'hello'; // Error：不能把 'string' 类型赋值给 'number' 类型
```

数组中：

```
const bar = [1, 2];
let [a, b] = bar;

a = 'hello'; // Error：不能把 'string' 类型赋值给 'number' 类型
```

如果函数参数能够被推断出来，那么解构亦是如此。在如下例子中，函数参数能够被解构为 `a/b` 成员：

```
type Adder = (number: { a: number; b: number }) => number;
function iTakeAnAdder(adder: Adder) {
  return adder({ a: 1, b: 2 });
}

iTakeAnAdder(({ a, b }) => {
  // a, b 的类型能被推断出来
  a = 'hello'; // Error：不能把 'string' 类型赋值给 'number' 类型
  return a + b;
});
```

类型保护

在前面章节类型保护中，我们已经知道它如何帮助我们改变和缩小类型范围（特别实在联合类型下）。类型保护只是一个块中变量另一种推断形式。

警告

小心使用参数

如果类型不能被赋值推断出来，类型也将不会流入函数参数中。例如如下的一个例子，编译器并不知道 `foo` 的类型，所它也就不能推断出 `a` 或者 `b` 的类型。

```
const foo = (a, b) => {
  /* do something */
};
```

然而，如果 `foo` 添加了类型注解，函数参数也就能被推断（ `a` ， `b` 都能被推断为 `number` 类型）：

```
type TwoNumberFunction = (a: number, b: number) => void;
const foo: TwoNumberFunction = (a, b) => {
  /* do something */
};
```

小心使用返回值

尽管 TypeScript 一般情况下能推断函数的返回值，但是它可能并不是你想要的。例如如下的 `foo` 函数，它的返回值为 `any` ：

```
function foo(a: number, b: number) {
  return a + addOne(b);
}

// 一些使用 JavaScript 库的特殊函数
function addOne(a) {
  return a + 1;
}
```

这是因为返回值的类型被一个缺少类型定义的 `addOne` 函数所影响（ `a` 是 `any` ，所以 `addOne` 返回值为 `any` ， `foo` 的返回值是也是 `any` ）。

TIP

我发现最简单的方式是明确的写上函数返回值，毕竟这些注解是一个定理，而函数是注解的一个证据。

这里还有一些其他可以想象的情景，但是有一个好消息是有编译器选项 `noImplicitAny` 可以捕获这些 bug。

noImplicitAny

选项 `noImplicitAny` 用来告诉编译器，当无法推断一个变量时发出一个错误（或者只能推断为一个隐式的 `any` 类型），你可以：

- 通过显式添加 `:any` 的类型注解，来让它成为一个 `any` 类型；

- 通过一些更正确的类型注解来帮助 `TypeScript` 推断类型。

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/typeInference.html*

类型兼容性

类型兼容性用于确定一个类型是否能赋值给其他类型。

如 `string` 类型与 `number` 类型不兼容：

```
let str: string = 'Hello';
let num: number = 123;

str = num; // Error: 'number' 不能赋值给 'string'
num = str; // Error: 'string' 不能赋值给 'number'
```

安全性

TypeScript 类型系统设计比较方便，它允许你有一些不正确的行为。例如：任何类型都能被赋值给 `any`，这意味着告诉编译器你可以做任何你想做的事情：

```
const foo: any = 123;
foo = 'hello';

foo.toPrecision(3);
```

结构化

TypeScript 对象是一种结构类型，这意味着只要结构匹配，名称也就无关紧要了：

```
interface Point {
  x: number;
  y: number;
}

class Point2D {
  constructor(public x: number, public y: number) {}
}

let p: Point;

// ok, 因为是结构化的类型
p = new Point2D(1, 2);
```

这允许你动态创建对象（就好像你在 `vanilla JS` 中使用一样），并且它如果能被推断，该对象仍然具有安全性。

```
interface Point2D {
  x: number;
  y: number;
}

interface Point3D {
  x: number;
  y: number;
  z: number;
}

const point2D: Point2D = {
  x: 0,
  y: 0
};

const point2D: Point2D = { x: 0, y: 10 };
const point3D: Point3D = { x: 0, y: 10, z: 20 };
function iTakePoint2D(point: Point2D) {
  /* do something */
}

iTakePoint2D(point2D); // ok, 完全匹配
iTakePoint2D(point3D); // 额外的信息，没关系
iTakePoint2D({ x: 0 }); // Error: 没有 'y'
```

变体

对类型兼容性来说，变体是一个利于理解和重要的概念。

对一个简单类型 `Base` 和 `Child` 来说，如果 `Child` 是 `Base` 的子类，`Child` 的实例能被赋值给 `Base` 类型的变量。

TIP

这是多态性。

在由 `Base` 和 `Child` 组合的复杂类型的类型兼容性中，它取决于相同场景下的 `Base` 与

`Child` 的变体：

- 协变（Covariant）：只在同一个方向；
- 逆变（Contravariant）：只在相反的方向；
- 双向协变（Bivariant）：包括同一个方向和不同方向；
- 不变（Invariant）：如果类型不完全相同，则它们是不兼容的。

TIP

对于存在完全可变数据的健全的类型系统（如 JavaScript），`Invariant` 是一个唯一的有效可选属性，但是如我们说讨论的，便利性迫使我们作出一些不是很安全的选择。

关于协变和逆变的更多内容，请参考：协变与逆变。

函数

当你在比较两个函数时，这有一些你需要考虑到的事情。

返回类型

协变（Covariant）：返回类型必须包含足够的数据。

```
interface Point2D {
  x: number;
  y: number;
}
interface Point3D {
  x: number;
  y: number;
  z: number;
}

let iMakePoint2D = (): Point2D => ({ x: 0, y: 0 });
let iMakePoint3D = (): Point3D => ({ x: 0, y: 0, z: 0 });

iMakePoint2D = iMakePoint3D;
iMakePoint3D = iMakePoint2D; // ERROR: Point2D 不能赋值给 Point3D
```

参数数量

更少的参数数量是好的（如：函数能够选择性的忽略一些多余的参数），但是你得保证有足够的参数被使用了：

```
const iTakeSomethingAndPassItAnErr = (x: (err: Error, data: any) => void) => {
  /* 做一些其他的 */
};

iTakeSomethingAndPassItAnErr(() => null); // ok
iTakeSomethingAndPassItAnErr(err => null); // ok
iTakeSomethingAndPassItAnErr((err, data) => null); // ok

// Error: 参数类型 `(err: any, data: any, more: any) => null` 不能赋值给参数类型 `(err: Error, data: any) => void`
iTakeSomethingAndPassItAnErr((err, data, more) => null);
```

可选的和 rest 参数

可选的（预先确定的）和 Rest 参数（任何数量的参数）都是兼容的:

```
let foo = (x: number, y: number) => {};
let bar = (x?: number, y?: number) => {};
let bas = (...args: number[]) => {};

foo = bar = bas;
bas = bar = foo;
```

Note

可选的（上例子中的 `bar`）与不可选的（上例子中的 `foo`）仅在选项为 `strictNullChecks` 为 `false` 时兼容。

函数参数类型

双向协变（Bivariant）: 旨在支持常见的事件处理方案。

```
// 事件等级
interface Event {
  timestamp: number;
}
interface MouseEvent extends Event {
  x: number;
  y: number;
}
interface KeyEvent extends Event {
```

```
  keyCode: number;
}

// 简单的事件监听
enum EventType {
  Mouse,
  Keyboard
}
function addEventListener(eventType: EventType, handler: (n: Event) => void) {
  // ...
}

// 不安全，但是有用，常见。函数参数的比较是双向协变。
addEventListener(EventType.Mouse, (e: MouseEvent) => console.log(e.x + ',' + e.y));

// 在安全情景下的一种不好方案
addEventListener(EventType.Mouse, (e: Event) => console.log((<MouseEvent>e).x + ',' + (<MouseEvent>e).y));
addEventListener(EventType.Mouse, <(e: Event) => void>((e: MouseEvent) => console.log(e.x + ',' + e.y)));

// 仍然不允许明确的错误，对完全不兼容的类型会强制检查
addEventListener(EventType.Mouse, (e: number) => console.log(e));
```

同样的，你也可以把 `Array<Child>` 赋值给 `Array<Base>` （协变），因为函数是兼容的。数组的协变需要所有的函数 `Array<Child>` 都能赋值给 `Array<Base>` ，例如 `push(t: Child)` 能被赋值给 `push(t: Base)` ，这都可以通过函数参数双向协变实现。

这对于来自其他语言的人来说，可能会感到很困惑，但他们希望以下错误不会出现在 TypeScript 中：

```
interface Poin2D {
  x: number;
  y: number;
}

let iTakePoint2D = (point: Point2D) => {};
let iTakePoint3D = (point: Point3D) => {};

iTakePoint3D = iTakePoint2D; // ok, 这是合理的
iTakePoint2D = iTakePoint3D; // ok, 为什么？
```

枚举

- 枚举与数字类型相互兼容

```
enum Status {
  Ready,
  Waiting
}

let status = Status.Ready;
let num = 0;

status = num;
num = status;
```

- 来自于不同枚举的枚举变量，被认为是不兼容的：

```
enum Status {
  Ready,
  Waiting
}
enum Color {
  Red,
  Blue,
  Green
}

let status = Status.Ready;
let color = Color.Red;

status = color; // Error
```

类

- 仅仅只有实例成员和方法会相比较，构造函数和静态成员不会被检查。

```
class Animal {
  feet: number;
  constructor(name: string, numFeet: number) {}
}

```

```
class Size {
  feet: number;
  constructor(meters: number) {}
}

let a: Animal;
let s: Size;

a = s; // OK
s = a; // OK
```

- 私有的和受保护的成员必须来自于相同的类。

```
class Animal {
  protected feet: number;
}
class Cat extends Animal {}

let animal: Animal;
let cat: Cat;

animal = cat; // ok
cat = animal; // ok

class Size {
  protected feet: number;
}

let size: Size;

animal = size; // ERROR
size = animal; // ERROR
```

泛型

TypeScript 类型系统基于变量的结构，仅当类型参数在被一个成员使用时，才会影响兼容性。如下例子中， `T` 对兼容性没有影响：

```
interface Empty<T> {}

let x: Empty<number>;
```

```
let y: Empty<string>;

x = y; // ok
```

当 `T` 被成员使用时，它将在实例化泛型后影响兼容性：

```
interface Empty<T> {
  data: T;
}

let x: Empty<number>;
let y: Empty<string>;

x = y; // Error
```

如果尚未实例化泛型参数，则在检查兼容性之前将其替换为 `any` ：

```
let identity = function<T>(x: T): T {
  // ...
};

let reverse = function<U>(y: U): U {
  // ...
};

identity = reverse; // ok, 因为 `(x: any) => any` 匹配 `(y: any) => any`
```

类中的泛型兼容性与前文所提及一致：

```
class List<T> {
  add(val: T) {}
}

class Animal {
  name: string;
}
class Cat extends Animal {
  meow() {
    // ..
  }
}

```

```
const animals = new List<Animal>();
animals.add(new Animal()); // ok
animals.add(new Cat()); // ok

const cats = new List<Cat>();
cats.add(new Animal()); // Error
cats.add(new Cat()); // ok
```

脚注：不变性（Invariance）

我们说过，不变性可能是唯一一个听起来合理的选项，这里有一个关于 `contra` 和 `co` 的变体，被认为对数组是不安全的。

```
class Animal {
  constructor(public name: string) {}
}
class Cat extends Animal {
  meow() {
    console.log('cat');
  }
}

let animal = new Animal('animal');
let cat = new Cat('cat');

// 多态
// Animal <= Cat

animal = cat; // ok
cat = animal; // ERROR: cat 继承于 animal

// 演示每个数组形式
let animalArr: Animal[] = [animal];
let catArr: Cat[] = [cat];

// 明显的坏处，逆变
// Animal <= Cat
// Animal[] >= Cat[]
catArr = animalArr; // ok, 如有有逆变
catArr[0].meow(); // 允许，但是会在运行时报错

```

```
// 另外一个坏处，协变
// Animal <= Cat
// Animal[] <= Cat[]
animalArr = catArr; // ok, 协变

animalArr.push(new Animal('another animal')); // 仅仅是 push 一个 animal 至 carArr 里
catArr.forEach(c => c.meow()); // 允许，但是会在运行时报错。
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/typeCompatibility.html*

Never

TIP

一个关于 never 的介绍视频

程序语言的设计确实应该存在一个底部类型的概念，当你在分析代码流的时候，这会是一个理所当然存在的类型。TypeScript 就是这样一种分析代码流的语言（？），因此它需要一个可靠的，代表永远不会发生的类型。

`never` 类型是 TypeScript 中的底层类型。它自然被分配的一些例子：

- 一个从来不会有返回值的函数（如：如果函数内含有 `while(true) {}`）；
- 一个总是会抛出错误的函数（如：`function foo() { throw new Error('Not Implemented') }`，`foo` 的返回类型是 `never`）；
  你也可以将它用做类型注解：

```
let foo: never; // ok
```

但是，`never` 类型仅能被赋值给另外一个 `never`：

```
let foo: never = 123; // Error: number 类型不能赋值给 never 类型

// ok, 做为函数返回类型的 never
let bar: never = (() => {
  throw new Error('Throw my hands in the air like I just dont care');
})();
```

很棒，现在让我们看看它的关键用例。

用例：详细的检查

```
function foo(x: string | number): boolean {
  if (typeof x === 'string') {
    return true;
  } else if (typeof x === 'number') {
    return false;
  }

  // 如果不是一个 never 类型，这会报错：
```

```
  // - 不是所有条件都有返回值 （严格模式下）
  // - 或者检查到无法访问的代码
  // 但是由于 TypeScript 理解 `fail` 函数返回为 `never` 类型
  // 它可以让你调用它，因为你可能会在运行时用它来做安全或者详细的检查。
  return fail('Unexhaustive');
}

function fail(message: string): never {
  throw new Error(message);
}
```

`never` 仅能被赋值给另外一个 `never` 类型，因此你可以用它来进行编译时的全面的检查，我们将会在辨析联合类型中讲解它。

与 void 的差异

一旦有人告诉你，`never` 表示一个从来不会优雅的返回的函数时，你可能马上就会想到与此类似的 `void`，然而实际上，`void` 表示没有任何类型，`never` 表示永远不存在的值的类型。

当一个函数没有返回值时，它返回了一个 `void` 类型，但是，当一个函数根本就没有返回值时（或者总是抛出错误），它返回了一个 `never`，`void` 指可以被赋值的类型（在 `strictNullChecking` 为 false 时），但是 `never` 不能赋值给其他任何类型，除了 `never`。

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/neverType.html*

辨析联合类型

当类中含有字面量成员时，我们可以用该类的属性来辨析联合类型。

做为一个例子，考虑 `Square` 和 `Rectangle` 的联合类型 `Shape`。`Square` 和 `Rectangle` 有共同成员 `kind`，因此 `kind` 存在于 `Shape` 中。

```
interface Square {
  kind: 'square';
  size: number;
}

interface Rectangle {
  kind: 'rectangle';
  width: number;
  height: number;
}

type Shape = Square | Rectangle;
```

如果你使用类型保护风格的检查（`==`、`===`、`!=`、`!==`）或者使用具有判断性的属性（在这里是 `kind`），TypeScript 将会认为你会使用的对象类型一定是拥有特殊字面量的，并且它会为你自动把类型范围变小：

```
function area(s: Shape) {
  if (s.kind === 'square') {
    // 现在 TypeScript 知道 s 的类型是 Square
    // 所以你现在能安全使用它
    return s.size * s.size;
  } else {
    // 不是一个 square ？因此 TypeScript 将会推算出 s 一定是 Rectangle
    return s.width * s.height;
  }
}
```

详细的检查

通常，联合类型的成员有一些自己的行为（代码）：

```
interface Square {
  kind: 'square';
  size: number;
}

interface Rectangle {
  kind: 'rectangle';
  width: number;
  height: number;
}

// 有人仅仅是添加了 `Circle` 类型
// 我们可能希望 TypeScript 能在任何被需要的地方抛出错误
interface Circle {
  kind: 'circle';
  radius: number;
}

type Shape = Square | Rectangle | Circle;
```

一个可能会让你的代码变差的例子：

```
function area(s: Shape) {
  if (s.kind === 'square') {
    return s.size * s.size;
  } else if (s.kind === 'rectangle') {
    return s.width * s.height;
  }

  // 如果你能让 TypeScript 给你一个错误，这是不是很棒？
}
```

你可以通过一个简单的向下思想，来确保块中的类型被推断为与 `never` 类型兼容的类型。例如，你可以添加一个更详细的检查来捕获错误：

```
function area(s: Shape) {
  if (s.kind === 'square') {
    return s.size * s.size;
  } else if (s.kind === 'rectangle') {
    return s.width * s.height;
  } else {
```

```
    // Error: 'Circle' 不能被赋值给 'never'
    const _exhaustiveCheck: never = s;
  }
}
```

它将强制你添加一种新的条件：

```
function area(s: Shape) {
  if (s.kind === 'square') {
    return s.size * s.size;
  } else if (s.kind === 'rectangle') {
    return s.width * s.height;
  } else if (s.kind === 'circle') {
    return Math.PI * s.radius ** 2;
  } else {
    // ok
    const _exhaustiveCheck: never = s;
  }
}
```

Switch

TIP

你可以通过 `switch` 来实现以上例子。

```
function area(s: Shape) {
  switch (s.kind) {
    case 'square':
      return s.size * s.size;
    case 'rectangle':
      return s.width * s.height;
    case 'circle':
      return Math.PI * s.radius ** 2;
    default:
      const _exhaustiveCheck: never = s;
  }
}
```

strictNullChecks

如果你使用 `strictNullChecks` 选项来做详细的检查，你应该返回 `_exhaustiveCheck` 变量（类型是 `never` ），否则 TypeScript 可能会推断返回值为 `undefined` ：

```
function area(s: Shape) {
  switch (s.kind) {
    case 'square':
      return s.size * s.size;
    case 'rectangle':
      return s.width * s.height;
    case 'circle':
      return Math.PI * s.radius ** 2;
    default:
      const _exhaustiveCheck: never = s;
      return _exhaustiveCheck;
  }
}
```

Redux

Redux 库正是使用的上述例子。

以下是添加了 TypeScript 类型注解的redux 要点。

```
import { createStore } from 'redux';

type Action =
  | {
      type: 'INCREMENT';
    }
  | {
      type: 'DECREMENT';
    };

/**
 * This is a reducer, a pure function with (state, action) => state signature.
 * It describes how an action transforms the state into the next state.
 *
 * The shape of the state is up to you: it can be a primitive, an array, an object,
 * or even an Immutable.js data structure. The only important part is that you should
```

```
 * not mutate the state object, but return a new object if the state changes.
 *
 * In this example, we use a `switch` statement and strings, but you can use a helper that
 * follows a different convention (such as function maps) if it makes sense for your
 * project.
 */
function counter(state = 0, action: Action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}

// Create a Redux store holding the state of your app.
// Its API is { subscribe, dispatch, getState }.
let store = createStore(counter);

// You can use subscribe() to update the UI in response to state changes.
// Normally you'd use a view binding library (e.g. React Redux) rather than subscribe() directly.
// However it can also be handy to persist the current state in the localStorage.

store.subscribe(() => console.log(store.getState()));

// The only way to mutate the internal state is to dispatch an action.
// The actions can be serialized, logged or stored and later replayed.
store.dispatch({ type: 'INCREMENT' });
// 1
store.dispatch({ type: 'INCREMENT' });
// 2
store.dispatch({ type: 'DECREMENT' });
// 1
```

与 TypeScript 一起使用可以有效的防止拼写错误，并且能提高重构和书写文档化代码的能力。

> 原文: *https://jkchao.github.io/typescript-book-*

> *chinese/typings/discrominatedUnion.html*

索引签名

可以用字符串访问 JavaScript 中的对象（TypeScript 中也一样），用来保存对其他对象的引用。

例如：

```
let foo: any = {};
foo['Hello'] = 'World';
console.log(foo['Hello']); // World
```

我们在键 `Hello` 下保存了一个字符串 `World`，除字符串外，它也可以保存任意的 JavaScript 对象，例如一个类的实例。

```
class Foo {
  constructor(public message: string) {}
  log() {
    console.log(this.message);
  }
}

let foo: any = {};
foo['Hello'] = new Foo('World');
foo['Hello'].log(); // World
```

当你传入一个其他对象至索引签名时，JavaScript 会在得到结果之前会先调用 `.toString` 方法：

```
let obj = {
  toString() {
    console.log('toString called');
    return 'Hello';
  }
};

let foo: any = {};
foo[obj] = 'World'; // toString called
console.log(foo[obj]); // toString called, World
console.log(foo['Hello']); // World
```

TIP

只要索引位置使用了 `obj` ， `toString` 方法都将会被调用。

数组有点稍微不同，对于一个 `number` 类型的索引签名，JavaScript 引擎将会尝试去优化（这取决于它是否是一个真的数组、存储的项目结构是否匹配等）。因此， `number` 应该被考虑作为一个有效的对象访问器（这与 `string` 不同），如下例子：

```
let foo = ['World'];
console.log(foo[0]); // World
```

因此，这就是 JavaScript。现在让我们看看 TypeScript 对这些概念更优雅的处理。

TypeScript 索引签名

JavaScript 在一个对象类型的索引签名上会隐式调用 `toString` 方法，而在 TypeScript 中，为防止初学者砸伤自己的脚（我总是看到 stackoverflow 上有很多 JavaScript 使用者都会这样。），它将会抛出一个错误。

```
const obj = {
  toString() {
    return 'Hello';
  }
};

const foo: any = {};

// ERROR: 索引签名必须为 string, number....
foo[obj] = 'World';

// FIX: TypeScript 强制你必须明确这么做：
foo[obj.toString()] = 'World';
```

强制用户必须明确的写出 `toString()` 的原因是：在对象上默认执行的 `toString` 方法是有害的。例如 v8 引擎上总是会返回 `[object Object]`

```
const obj = { message: 'Hello' };
let foo: any = {};

// ERROR: 索引签名必须为 string, number....
foo[obj] = 'World';
```

```
// 这里实际上就是你存储的地方
console.log(foo['[object Object]']); // World
```

当然，数字类型是被允许的，这是因为：

- 需要数组 / 元组的支持；
- 即使你在数组里使用一个 `obj`，这个默认被调用的 `toString` 方法，被实现的很好（不是 `[object Object]`）。
  如下所示：

```
console.log((1).toString()); // 1
console.log((2).toString()); // 2
```

因此，我们有以下结论：

TIP

TypeScript 的索引签名必须是 `string` 或者 `number`。

`symbols` 也是有效的，TypeScript 支持它。在接下来我们将会讲解它。

声明一个索引签名

在上文中，我们通过使用 `any` 来让 TypeScript 允许我们可以做任意我们想做的事情。实际上，我们可以明确的指定索引签名。例如：假设你想确认存储在对象中任何内容都符合 `{ message: string }` 的结构，你可以通过 `[index: string]: { message: string }` 来实现。

```
const foo: {
  [index: string]: { message: string };
} = {};

// 储存的东西必须符合结构
// ok
foo['a'] = { message: 'some message' };

// Error, 必须包含 `message`
foo['a'] = { messages: 'some message' };

// 读取时，也会有类型检查
// ok
foo['a'].message;
```

```
// Error: messages 不存在
foo['a'].messages;
```

TIP

索引签名的名称（如：`{ [index: string]: { message: string } }` 里的 `index` ）除了可读性外，并没有任何意义。例如：如果有一个用户名，你可以使用 `{ username: string}: { message: string }`，这有利于下一个开发者理解你的代码。

`number` 类型的索引也支持：`{ [count: number]: 'SomeOtherTypeYouWantToStoreEgRebate' }`。

所有成员都必须符合字符串的索引签名

当你声明一个索引签名时，所有明确的成员都必须符合索引签名：

```
// ok
interface Foo {
  [key: string]: number;
  x: number;
  y: number;
}

// Error
interface Bar {
  [key: string]: number;
  x: number;
  y: string; // Error: y 属性必须为 number 类型
}
```

这可以给你提供安全性，任何以字符串的访问都能得到相同结果。

```
interface Foo {
  [key: string]: number;
  x: number;
}

let foo: Foo = {
  x: 1,
  y: 2
};
```

```
// 直接
foo['x']; // number

// 间接
const x = 'x';
foo[x]; // number
```

使用一组有限的字符串字面量

一个索引签名可以通过映射类型来使索引字符串为联合类型中的一员，如下所示：

```
type Index = 'a' | 'b' | 'c';
type FromIndex = { [k in Index]?: number };

const good: FromIndex = { b: 1, c: 2 };

// Error:
// `{ b: 1, c: 2, d: 3 }` 不能分配给 'FromIndex'
// 对象字面量只能指定已知类型，'d' 不存在 'FromIndex' 类型上
const bad: FromIndex = { b: 1, c: 2, d: 3 };
```

这通常与 `keyof/typeof` 一起使用，来获取变量的类型，在下一章节中，我们将解释它。

变量的规则一般可以延迟被推断：

```
type FromSomeIndex<K extends string> = { [key in K]: number };
```

同时拥有 string 和 number 类型的索引签名

这并不是一个常见的用例，但是 TypeScript 支持它。

`string` 类型的索引签名比 `number` 类型的索引签名更严格。这是故意设计，它允许你有如下类型：

```
interface ArrStr {
  [key: string]: string | number; // 必须包括所用成员类型
  [index: number]: string; // 字符串索引类型的子级

  // example
```

```
  length: number;
}
```

设计模式：索引签名的嵌套

TIP

添加索引签名时，需要考虑的 API。

在 JavaScript 社区你将会见到很多滥用索引签名的 API。如 JavaScript 库中使用 CSS 的常见模式：

```
interface NestedCSS {
  color?: string; // strictNullChecks=false 时索引签名可为 undefined
  [selector: string]: string | NestedCSS;
}

const example: NestedCSS = {
  color: 'red',
  '.subclass': {
    color: 'blue'
  }
};
```

尽量不要使用这种把字符串索引签名与有效变量混合使用。如果属性名称中有拼写错误，这个错误不会被捕获到：

```
const failsSilently: NestedCSS = {
  colour: 'red' // 'colour' 不会被捕捉到错误
};
```

取而代之，我们把索引签名分离到自己的属性里，如命名为 `nest`（或者 `children`、`subnodes` 等）：

```
interface NestedCSS {
  color?: string;
  nest?: {
    [selector: string]: NestedCSS;
  };
}

```

```
const example: NestedCSS = {
  color: 'red',
  nest: {
    '.subclass': {
      color: 'blue'
    }
  }
}

const failsSliently: NestedCSS {
  colour: 'red'  // TS Error: 未知属性 'colour'
}
```

索引签名中排除某些属性

有时，你需要把属性合并至索引签名（虽然我们并不建议这么做，你应该使用上文中提到的嵌套索引签名的形式），如下例子：

```
type FieldState = {
  value: string;
};

type FromState = {
  isValid: boolean; // Error: 不符合索引签名
  [filedName: string]: FieldState;
};
```

TypeScript 会报错，因为添加的索引签名，并不兼容它原有的类型，使用交叉类型可以解决上述问题：

```
type FieldState = {
  value: string;
};

type FormState = { isValid: boolean } & { [fieldName: string]: FieldState };
```

请注意尽管你可以声明它至一个已存在的 TypeScript 类型上，但是你不能创建如下的对象：

```
type FieldState = {
  value: string;
};
```

```

type FormState = { isValid: boolean } & { [fieldName: string]: FieldState };

// 将它用于从某些地方获取的 JavaScript 对象
declare const foo: FormState;

const isValidBool = foo.isValid;
const somethingFieldState = foo['something'];

// 使用它来创建一个对象时，将不会工作
const bar: FormState = {
  // 'isValid' 不能赋值给 'FieldState'
  isValid: false
};
```

> **原文**: *https://jkchao.github.io/typescript-book-chinese/typings/indexSignatures.html*

流动的类型

TypeScript 类型系统非常强大，它支持其他任何单一语言无法实现的类型流动和类型片段。

这是因为 TypeScript 的设计目的之一是让你无缝与像 JavaScript 这类高动态的语言一起工作。在这里，我们介绍一些在 TypeScript 中使用移动类型的技巧。

关键的动机：当你改变了其中一个时，其他相关的会自动更新，并且当有事情变糟糕时，你会得到一个有好的提示，就好像一个被精心设计过的约束系统。

复制类型和值

如果你想移动一个类，你可能会想要做以下事情：

```
class Foo {}

const Bar = Foo;

let bar: Bar; // Error: 不能找到名称 'Bar'
```

这会得到一个错误，因为 `const` 仅仅是复制了 `Foo` 到一个变量声明空间，因此你无法把 `Bar` 当作一个类型声明使用。正确的方式是使用 `import` 关键字，请注意，如果你在使用 `namespace` 或者 `modules`，使用 `import` 是你唯一能用的方式：

```
namespace importing {
  export class Foo {}
}

import Bar = importing.Foo;
let bar: Bar; // ok
```

这个 `import` 技巧，仅适合于类型和变量。

捕获变量的类型

你可以通过 `typeof` 操作符在类型注解中使用变量。这允许你告诉编译器，一个变量的类型与其他类型相同，如下所示：

```
let foo = 123;
```

```
let bar: typeof foo; // 'bar' 类型与 'foo' 类型相同（在这里是： 'number'）

bar = 456; // ok
bar = '789'; // Error: 'string' 不能分配给 'number' 类型
```

捕获类成员的类型

与捕获变量的类型相似，你仅仅是需要声明一个变量用来捕获到的类型：

```
class Foo {
  foo: number; // 我们想要捕获的类型
}

declare let _foo: Foo;

// 与之前做法相同
let bar: typeof _foo.foo;
```

捕获字符串类型

许多 JavaScript 库和框架都使用原始的 JavaScript 字符串，你可以使用 `const` 定义一个变量捕获它的类型：

```
// 捕获字符串的类型与值
const foo = 'Hello World';

// 使用一个捕获的类型
let bar: typeof foo;

// bar 仅能被赋值 'Hello World'
bar = 'Hello World'; // ok
bar = 'anything else'; // Error
```

在这个例子里， `bar` 有字面量类型 `Hello World` ，我们在字面量类型章节已经深入讨论。

捕获键的名称

`keyof` 操作符能让你捕获一个类型的键。例如，你可以使用它来捕获变量的键名称，在通过使用 `typeof` 来获取类型之后：

```
const colors = {
  red: 'red',
  blue: 'blue'
};

type Colors = keyof typeof colors;

let color: Colors; // color 的类型是 'red' | 'blue'
color = 'red'; // ok
color = 'blue'; // ok
color = 'anythingElse'; // Error
```

这允许你很容易地拥有像字符串枚举+常量这样的类型，如上例所示。

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/movingTypes.html*

异常处理

JavaScript 有一个 `Error` 类，用于处理异常。你可以通过 `throw` 关键字来抛出一个错误。然后通过 `try/catch` 块来捕获此错误：

```
try {
  throw new Error('Something bad happened');
} catch (e) {
  console.log(e);
}
```

错误子类型

除内置的 `Error` 类外，还有一些额外的内置错误，它们继承自 `Error` 类：

RangeError

当数字类型变量或者参数超出其有效范围时，出现 `RangeError` 的错误提示：

```
// 使用过多参数调用 console
console.log.apply(console, new Array(1000000000)); // RangeError: 数组长度无效
```

ReferenceError

当引用无效时，会出现 `ReferenceError` 的错误提示：

```
'use strict';
console.log(notValidVar); // ReferenceError: notValidVar 未定义
```

SyntaxError

当解析无效 JavaScript 代码时，会出现 `SyntaxError` 的错误提示：

```
1 *** 3   // SyntaxError: 无效的标记 *
```

TypeError

变量或者参数不是有效类型时，会出现 `TypeError` 的错误提示：

```
'1.2'.toPrecision(1); // TypeError: '1.2'.toPrecision 不是函数。
```

URIError

当传入无效参数至 `encodeURI()` 和 `decodeURI()` 时，会出现 `URIError` 的错误提示：

```
decodeURI('%'); // URIError: URL 异常
```

使用 Error

JavaScript 初学者可能有时候仅仅是抛出一个原始字符串：

```
try {
  throw 'Something bad happened';
} catch (e) {
  console.log(e);
}
```

不要这么做，使用 `Error` 对象的基本好处是，它能自动跟踪堆栈的属性构建以及生成位置。

原始字符串会导致极差的调试体验，并且在分析日志时，将会变得错综复杂。

你并不需要 throw 抛出一个错误

传递一个 `Error` 对象是没问题的，这种方式在 `Node.js` 回调函数中非常常见，它用第一个参数作为错误对象进行回调处理。

```
function myFunction (callback: (e: Error)) {
  doSomethingAsync(function () {
    if (somethingWrong) {
      callback(new Error('This is my error'));
    } else {
      callback();
    }
  })
}
```

优秀的用例

「Exceptions should be exceptional」是计算机科学中常用用语。这里有一些原因说明在 JavaScript(TypeScript) 中也是如此。

不清楚从哪里抛出错误

考虑如下代码块：

```
try {
  const foo = runTask1();
  const bar = runTask2();
} catch (e) {
  console.log('Error:', e);
}
```

下一个开发者可能并不清楚哪个函数可能会抛出错误。在没有阅读 `task1/task2` 代码以及他们可能会调用的函数时，对代码 `review` 的人员可能也不会知道错误会从哪里抛出。

优雅的捕获错误

你可以通过为每个可能抛出错误的代码显式捕获，来使其优雅：

```
try {
  const foo = runTask1();
} catch (e) {
  console.log('Error:', e);
}

try {
  const bar = runTask2();
} catch (e) {
  console.log('Error:', e);
}
```

但是现在，如果你想从第一个任务中传递变量到第二个任务中，代码会变的混乱（注意：foo 变量需要用 let 显式注解它，因为它不能从 `runTask1` 中返回出来）：

```
let foo: number; // Notice 使用 let 并且显式注名类型注解

try {
  foo = runTask1();
} catch (e) {
  console.log('Error:', e);
```

```
}

try {
  const bar = runTask2();
} catch (e) {
  console.log('Error:', e);
}
```

没有在类型系统中很好的表示

考虑如下函数:

```
function validate(value: number) {
  if (value < 0 || value > 100) {
    throw new Error('Invalid value');
  }
}
```

在这种情境下使用 `Error` 不是一个好的主意。因为没有用来验证函数的类型定义（如: `(value: number) => void`），取而代之一个更好的方式是创建一个验证方法:

```
function validate(
  value: number;
): {
  error?: string;
} {
  if (value < 0 || value > 100) {
    return { error: 'Invalid value' };
  }
}
```

现在它具有类型定义了。

TIP

除非你想用以非常通用（try/catch）的方式处理错误，否则不要抛出错误。

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/exceptionsHanding.html*

混合

TypeScript (和 JavaScript) 类只能严格的单继承，因此你不能做:

```
class User extends Tagged, Timestamped { // ERROR : 不能多重继承
  // ..
}
```

从可重用组件构建类的另一种方式是通过基类来构建它们，这种方式称为混合。

这个主意是简单的，采用函数 B 接受一个类 A，并且返回一个带有新功能的类的方式来替代 A 类扩展 B 来获取 B 上的功能，前者中的 B 即是混合。

TIP

「混合」是一个函数:

- 传入一个构造函数;
- 创建一个带有新功能，并且扩展构造函数的新类;
- 返回这个新类。

一个完整的例子:

```
// 所有 mixins 都需要
type Constructor<T = {}> = new (...args: any[]) => T;

/////////////
// mixins 例子
////////////

// 添加属性的混合例子
function TimesTamped<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    timestamp = Date.now();
  };
}

// 添加属性和方法的混合例子
function Activatable<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    isActivated = false;

```

```
    activate() {
      this.isActivated = true;
    }

    deactivate() {
      this.isActivated = false;
    }
  };
}

///////////
// 组合类
///////////

// 简答的类
class User {
  name = '';
}

// 添加 TimesTamped 的 User
const TimestampedUser = TimesTamped(User);

// Tina TimesTamped 和 Activatable 的类
const TimestampedActivatableUser = TimesTamped(Activatable(User));

//////////
// 使用组合类
//////////

const timestampedUserExample = new TimestampedUser();
console.log(timestampedUserExample.timestamp);

const timestampedActivatableUserExample = new TimestampedActivatableUser();
console.log(timestampedActivatableUserExample.timestamp);
console.log(timestampedActivatableUserExample.isActivated);
```

让我们分解这个例子。

创建一个构造函数

混合接受一个类，并且使用新功能扩展它。因此，我们需要定义构造函数的类型：

```
type Constructor<T = {}> = new (...args: any[]) => T;
```

扩展一个类并且返回它

```
// 添加属性的混合例子
function TimesTamped<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    timestamp = Date.now();
  };
}
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/mixins.html*

ThisType

通过 `ThisType` 我们可以在对象字面量中键入 `this`，并提供通过上下文类型控制 `this` 类型的便捷方式。它只有在 `—noImplicitThis` 的选项下才有效。

现在，在对象字面量方法中的 `this` 类型，将由以下决定：

- 如果这个方法显示指定了 `this` 参数，那么 `this` 具有该参数的类型。（下例子中 `bar` ）
- 否则，如果方法由带 `this` 参数的签名进行上下文键入，那么 `this` 具有该参数的类型。（下例子中 `foo` ）
- 否则，如果 `—noImplicitThis` 选项已经启用，并且对象字面量中包含由 `ThisType<T>` 键入的上下文类型，那么 `this` 的类型为 `T`。
- 否则，如果 `—noImplicitThis` 选项已经启用，并且对象字面量中不包含由 `ThisType<T>` 键入的上下文类型，那么 `this` 的类型为该上下文类型。
- 否则，如果 `—noImplicitThis` 选项已经启用，`this` 具有该对象字面量的类型。
- 否则，`this` 的类型为 `any`。
  一些例子：

```
// Compile with --noImplicitThis

type Point = {
  x: number;
  y: number;
  moveBy(dx: number, dy: number): void;
};

let p: Point = {
  x: 10,
  y: 20,
  moveBy(dx, dy) {
    this.x += dx; // this has type Point
    this.y += dy; // this has type Point
  }
};

let foo = {
  x: 'hello',
  f(n: number) {
    this; // { x: string, f(n: number): void }
```

```
  }
};

let bar = {
  x: 'hello',
  f(this: { message: string }) {
    this; // { message: string }
  }
};
```

类似的方式，当使用 `—noImplicitThis` 时，函数表达式赋值给 `obj.xxx` 或者 `obj[xxx]` 的目标时，在函数中 `this` 的类型将会是 `obj` ：

```
// Compile with --noImplicitThis

obj.f = function(n) {
  return this.x - n; // 'this' has same type as 'obj'
};

obj['f'] = function(n) {
  return this.x - n; // 'this' has same type as 'obj'
};
```

通过 API 转换参数的形式来生成 `this` 的值的情景下，可以通过创建一个新的 `ThisType<T>` 标记接口，可用于在上下文中表明转换后的类型。尤其是当字面量中的上下文类型为 `ThisType<T>` 或者是包含 `ThisType<T>` 的交集时，显得尤为有效，对象字面量方法中 `this` 的类型即为 `T` 。

```
// Compile with --noImplicitThis

type ObjectDescriptor<D, M> = {
  data?: D;
  methods?: M & ThisType<D & M>; // Type of 'this' in methods is D & M
};

function makeObject<D, M>(desc: ObjectDescriptor<D, M>): D & M {
  let data: object = desc.data || {};
  let methods: object = desc.methods || {};
  return { ...data, ...methods } as D & M;
}

let obj = makeObject({
```

```
  data: { x: 0, y: 0 },
  methods: {
    moveBy(dx: number, dy: number) {
      this.x += dx; // Strongly typed this
      this.y += dy; // Strongly typed this
    }
  }
});

obj.x = 10;
obj.y = 20;
obj.moveBy(5, 5);
```

在上面的例子中， `makeObject` 参数中的对象属性 `methods` 具有包含 `ThisType<D & M>` 的上下文类型，因此对象中 `methods` 属性下的方法的 `this` 类型为 `{ x: number, y: number } & { moveBy(dx: number, dy: number): number }`。

`ThisType<T>` 的接口，在 `lib.d.ts` 只是被声明为空的接口，除了可以在对象字面量上下文中可以被识别以外，该接口的作用等同于任意空接口。

> 原文: *https://jkchao.github.io/typescript-book-chinese/typings/thisType.html*

- 支持 JSX
- React JSX
- 非 React JSX

支持 JSX

TypeScript 支持 JSX 转换和代码分析，如果你还不了解 JSX，官网上有关于它的摘要：

> *JSX is an XML-like syntax extension to ECMAScript without any defined semantics. It's NOT intended to be implemented by engines or browsers. It's NOT a proposal to incorporate JSX into the ECMAScript spec itself. It's intended to be used by various preprocessors (transpilers) to transform these tokens into standard ECMAScript.*

JSX 背后的动机是允许用户在 JavaScript 中书写类似于 HTML 的视图，因此你可以：

- 使用相同代码，既能检查你的 JavaScript，同时能检查你的 HTML 视图层部分。
- 让视图层了解运行时的上下文（加强传统 MVC 中的控制器与视图连接）。
- 复用 JavaScript 设计模式维护 HTML 部分，例如：用 `Array.prototype.map.`、`?:`、`switch` 等，代替创建新的可替代品。

这能够减少错误的可能性，并且能增加用户界面的可维护性。目前 JSX 的主要消费者来自 facebook 推出的 ReactJS，接下来我们结合它来讨论 JSX 用法。

> 原文*: https://jkchao.github.io/typescript-book-chinese/jsx/support.html*

React JSX

> *在 React 中使用 TypeScript 的教学视频*

建立

在 TypeScript in the browser 章节中，我们已经学会开始开发 React 的应用了，以下是一些重点：

- 使用文件后缀 `.tsx`（替代 `.ts`）；
- 在你的 `tsconfig.json` 配置文件的 `compilerOptions` 里设置选项 `"jsx": "react"`；
- 在你的项目里为 `JSX` 和 `React` 安装声明文件：`npm i -D @types/react @types/react-dom`；
- 导入 `react` 到你的 `.tsx` 文件（`import * as React from 'react'`）。

HTML 标签 vs 组件

React 不但能渲染 HTML 标签（strings）也能渲染 React 组件（classes）。JavaScript 触发这些的原理是不同的（`React.createElement('div')` vs `React.createElement(MyComponent)`），确定使用哪一种方式取决于首字母的大小写，`foo` 被认为是 HTML 标签，`Foo` 被认为是一个组件。

类型检查

HTML 标签

一个 HTML 标签 `foo` 被标记为 `JSX.IntrinsicElements.foo` 类型。在我们已经安装的文件 `react-jsx.d.ts` 中定义了所有主要标签的类型，如下是一部分示例：

```
declare namespace JSX {
  interface IntrinsicElements {
    a: React.HTMLAttributes;
    abbr: React.HTMLAttributes;
    div: React.HTMLAttributes;
    span: React.HTMLAttributes;

    // 其他
  }
}
```

函数式组件

你可以使用 `React.FunctionComponent` 接口定义函数组件：

```
type Props = {
  foo: string;
};

const myComponent: React.FunctionComponent<Props> = props => {
  return <span>{props.foo}</span>;
};

<MyComponent foo="bar" />;
```

类组件

根据组件的 `props` 属性对组件进行类型检查。这是以 JSX 如何转换做为蓝本，例如：属性成为 `props` 的组成部分。

`react.d.ts` 文件定义了 `React.Component<Props,State>`，你应该使用自己所需的 `Props` 和 `State` 声明扩展它：

```
type Props = {
  foo: string;
};

class MyComponent extends React.Component<Props, {}> {
  render() {
    return <span>{this.props.foo}</span>;
  }
}

<MyComponent foo="bar" />;
```

React JSX Tip：接收组件的实例

react 类型声明文件提供了 `React.ReactElement<T>`，它可以让你通过传入 `<T/>`，来注解类组件的实例化结果。

```
class MyAwesomeComponent extends React.Component {
```

```
  render() {
    return <div>Hello</div>;
  }
}

const foo: React.ReactElement<MyAwesomeComponent> = <MyAwesomeComponent />; // Okay
const bar: React.ReactElement<MyAwesomeComponent> = <NotMyAwesomeComponent />; // Error!
```

TIP

当然，你可以将它用作函数参数的注解，甚至可以是 React 组件的 prop 成员。

React JSX Tip：接受一个可以在 Props 起作用，并使用 JSX 渲染的组件

类型 `React.Component<Props>` 是 `React.ComponentClass<P>` 与 `React.StatelessComponent<P>` 的组合，所以你可以接受一些可以用作 Props 类型和使用 JSX 渲染的组件。

```
const X: React.Component<Props> = foo; // from somewhere

// Render X with some props:
<X {...props} />;
```

React JSX Tip：可渲染的接口

React 可以渲染一些像 `JSX` 或者是 `string` 的内容，这些被合并到类型 `React.ReactNode` 中，因此，当你接收可渲染的内容时，你可以使用它：

```
type Props = {
  header: React.ReactNode;
  body: React.ReactNode;
};

class MyComonent extends React.Component<Props, {}> {
  render() {
    return (
      <div>
        {this.props.header}
```

```
      {this.props.body}
    </div>
  );
 }
}

<MyComponent foo="bar" />;
```

React JSX tip: 接收组件的接口

React 声明文件提供 `React.ReactElement<T>` 的接口，可以让你注解一个类组件实例化的返回值 `<T/>` ，如：

```
class MyAwesomeComponent extends React.Component {
  render() {
    return <div>Hello</div>;
  }
}

const foo: React.ReactElement<MyAwesomeComponent> = <MyAwesomeComponent />; // Okay
const bar: React.ReactElement<MyAwesomeComponent> = <NotMyAwesomeComponent />; // Error!
```

TIP

你也可以将其用做函数参数的注解，或者是 React 组件的 prop 注解。

React JSX tip: 接收可以做为 props 的组件，并且使用 JSX 渲染它

类型 `React.Component<Props>` 合并了 `React.ComponentClass<P>` 和 `React.StatelessComponent<P>` ，因此，你可以接收一些使用 `Prop` 类型的组件，并使用 JSX 渲染它：

```
const X: React.Component<Props> = foo // 来自其他地方

// 渲染 X
<X {...props} />
```

React JSX tip: 泛型组件

它完全能按我们预期工作，如：

```
// 一个泛型组件
type SelectProps<T> = { items: T[] };
class Select<T> extends React.Component<SelectProps<T>, any> {}

// 使用
const Form = () => <Select<string> items={['a', 'b']} />;
```

泛型函数

一些像下面这样的正常工作：

```
function foo<T>(x: T): T {
  return x;
}
```

然而不能使用箭头泛型函数：

```
const foo = <T>(x: T) => T; // Error: T 标签没有关闭
```

解决办法：在泛型参数里使用 `extends` 来提示编译器，这是个泛型：

```
const foo = <T extends {}>(x: T) => x;
```

React Tip: 强类型的 Refs

基本上你在初始化一个变量时，使用 `ref` 和 `null` 的联合类型，并且在回调函数中初始化他：

```
class Example extends React.Component {
  example() {
    // ... something
  }

  render() {
    return <div>Foo</div>;
  }
}

class Use {
  exampleRef: Example | null = null;
```

```
  render() {
    return <Example ref={exampleRef => (this.exampleRef = exampleRef)} />;
  }
}
```

使用原生元素时也一样：

```
class FocusingInput extends React.Component<{ value: string; onChange: (value: string) => any }, {}> {
  input: HTMLInputElement | null = null;

  render() {
    return (
      <input
        ref={input => (this.input = input)}
        value={this.props.value}
        onChange={e => {
          this.props.onChange(e.target.value);
        }}
      />
    );
  }
  focus() {
    if (this.input != null) {
      this.input.focus();
    }
  }
}
```

类型断言

如我们之前提到的，可以使用 `as Foo` 语法进行类型断言。

默认 Props

- 在有状态组件中使用默认的 Props：你可以通过 `null` 操作符（这不是一个理想的方式，但是这是我能想到的最简单的最小代码解决方案）告诉 TypeScript 一个属性将会被外部提供（React）。

```
class Hello extends React.Component<{
```

```
  /**
   * @default 'TypeScript'
   */
  compiler?: string;
  framework: string;
}> {
  static defaultProps = {
    compiler: 'TypeScript'
  };
  render() {
    const compiler = this.props.compiler!;
    return (
      <div>
        <div>{compiler}</div>
        <div>{this.props.framework}</div>
      </div>
    );
  }
}

ReactDOM.render(
  <Hello framework="React" />, // TypeScript React
  document.getElementById('root')
);
```

- 在 SFC 中使用默认的 Props：推荐使用简单的 JavaScript 参数，因为同样适用于 TypeScript 类型系统：

```
const Hello: React.SFC<{
  /**
   * @default 'TypeScript'
   */
  compiler?: string;
  framework: string;
}> = ({
  compiler = 'TypeScript', // Default prop
  framework
}) => {
  return (
    <div>
      <div>{compiler}</div>
      <div>{framework}</div>
```

```
    </div>
  );
};

ReactDOM.render(
  <Hello framework="React" />, // TypeScript React
  document.getElementById('root')
);
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/jsx/reactJSX.html*

非 React JSX

TypeScript 使你能够以类型安全的方式使用在 React 中使用 JSX 之外的其他方式，以下列出了一些可自定义点，但是请注意，这些适用于 UI 框架的作者：

- 你可以 `"jsx": "preserve"` 选项来禁用 `react` 样式触发，这意味着，JSX 将按原样触发，然后你可以使用自定义转化器来转化 JSX 部分；
- 使用 `JSX` 全局模块：
    - 你可以通过定制 `JSX.IntrinsicElements` 的接口成员来控制哪些 HTML 标签是可取的，以及如何对其进行类型检查；
    - 当你在组件中使用：
        - 你可以通过自定义默认的 `interface ElementClass extends React.Component<any, any> { }` 声明文件来控制哪个 `class` 必须由组件继承；
        - 你可以通过自定义 `declare module JSX { interface ElementAttributesProperty { props: {} } }` 声明文件来控制使用的哪个属性（property）来检查特性（attribute）（默认是 `props`）。

jsxFactory

通过 `–jsxFactory <JSX factory Name>` 与 `–jsx react`，能让你不同于默认 `React` 的方式使用 JSX 工厂函数。

这个新的工厂函数名字习惯被称之为 `createElement` 函数。

例子

```
import { jsxFactory } from 'jsxFactory';

const div = <div>Hello JSX!</div>;
```

使用编译：

```
tsc --jsx react --reactNamespace jsxFactory --m commonJS
```

编译结果：

```
'use strict';
var jsxFactory_1 = require('jsxFactory');
```

```
var div = jsxFactory_1.jsxFactory.createElement('div', null, 'Hello JSX!');
```

jsx 编译提示

你甚至可以使用 `jsxPragma` 为每个文件指定不同的 `jsxFactory`：

```
/** @jsx jsxFactory */
import { jsxFactory } from 'jsxFactory';

var div = <div>Hello JSX!</div>;
```

在 jsx 编译提示中，配合 `—jsx react` 命令，这个文件将会被触发使用工厂函数：

```
'use strict';
var jsxFactory_1 = require('jsxFactory');
var div = jsxFactory_1.jsxFactory.createElement('div', null, 'Hello JSX!');
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/jsx/nonReactJSX.html*

- 解读 Errors
- 常见的 Error

解读 Errors

TypeScript 是一种专注于帮助开发人员的编程语言，当错误出现时，它会提供尽可能提供非常有用的错误信息。这对于那些信任使用者的编译器来说，可能会导致轻微的信息量过载，而不会那么实用。

让我们来看一个在 IDE 中的例子：

```
type SomethingComplex = {
  foo: number;
  bar: string;
};

function takeSomethingComplex(arg: SomethingComplex) {}

function getBar(): string {
  return 'some bar';
}

// 一个可能会出现的错误使用
const fail = {
  foo: 123,
  bar: getBar
};

takeSomethingComplex(fail); // 在这里 TypeScript 会报错
```

这个简单的例子，演示了一个常见的程序设计错误，它调用函数失败（ `bar: getBar` 应该是 `bar: getBar()` ）。幸运的是，一旦不符合类型要求，TypeScript 将会捕捉到这个错误。

错误分类

TypeScript 错误信息分为两类：简洁和详细。

简洁

简洁的错误信息是为了提供一个编译器描述的错误号以及一些相关的信息，一个简洁的错误信息类似于如下所示：

```
TS2345: Argument of type '{ foo: number; bar: () => string; }' is not assignable to parameter of type 'SomethingComplex'.
```

然而，它没有提供更深层次的信息，如为什么这个错误会发生。这就是详细错误所需要的原因。

详细

详细的错误信息类似于如下所示：

```
[ts]
Argument of type '{ foo: number; bar: () => string; }' is not assignable to parameter of type 'SomethingComplex'.
  Types of property 'bar' are incompatible.
    Type '() => string' is not assignable to type 'string'.
```

详细的错误信息是为了指导使用者知道为什么一些错误（在这个例子里是类型不兼容）会发生。第一行与简洁的错误信息相同，后跟一些详细的信息。你应该阅读这些详细信息，因为对于开发者的一些疑问，它都给出了问答：

```
ERROR: Argument of type '{ foo: number; bar: () => string; }' is not assignable to parameter of type 'SomethingComplex'.

WHY?
CAUSE ERROR: Types of property 'bar' are incompatible.

WHY?
CAUSE ERROR: Type '() => string' is not assignable to type 'string'.
```

所以，最根本的原因是：

- 在属性 `bar`
- 函数 `() => string` 它应该是一个字符串。
  这能够帮助开发者修复 bar 属性的 bug（它们忘记了调用这个函数）。

在 IDE 中怎么提示

IDE 通常会在详细的错误提示之后显示简洁版本，如下所示：

```
[ts]
Argument of type '{ foo: number; bar: () => string; }' is
 not assignable to parameter of type 'SomethingComplex'.
  Types of property 'bar' are incompatible.
    Type '() => string' is not assignable to type 'strin
g'.
```

```
TS2345: Argument of type '{ foo: number; bar: () => strin
g; }' is not assignable to parameter of type 'SomethingCom
plex'.
```

- 你通常可能只会阅读「为什么」的详细信息；
- 当你想寻找相同的错误时（使用 `TSXXX` 错误编号，或者部分错误信息），使用简洁的版本。

> 原文: *https://jkchao.github.io/typescript-book-chinese/error/interpreting.html*

常见的 Error

在此章节中，我们学习在实际应用中将会遇到的常见错误代码。

TS2304

例子：

> `Cannot find name ga`, `Cannot find name $`, `Cannot find module jquery`

你可能在使用第三方的库（如：google analytics），但是你并没有 `declare` 的声明。在没有声明它们之前，TypeScript 试图避免错误和使用变量。因此在使用一些额外的库时，你需要明确的声明使用的任何变量（如何修复它）。

TS2307

例子：

> `Cannot find module 'underscore'`

你可能把第三方的库做为模块（移步模块）来使用，并且没有一个与之对应的环境声明文件（更多声明文件信息）。

TS1148

例子：

> `Cannot compile modules unless the '—module' flag provided`

请查看模块章节

捕获不能有类型注解的简短变量

例子：

```
try {
  something();
} catch (e) {
  // 捕获不能有类型注解的简短变量
  // ...
}
```

TypeScript 正在保护你免受 JavaScript 代码的侵害，取而代之，使用类型保护：

```
try {
  something();
} catch (e) {
  // 捕获不能有类型注解的简短变量
  if (e instanceof Error) {
    // do...
  }
}
```

接口 ElementClass 不能同时扩展类型别名 Component 和 Component

当在编译上下文中同时含有两个 `react.d.ts` （ `@types/react/index.d.ts` ）会发生这种情况。

修复：

- 删除 `node_modules` 和任何 `package-lock` （或者 `yarn lock` ），然后再一次 `npm install` ；
- 如果这不能工作，查找无效的模块（你所使用的所用用到了 `react.d.ts` 模块应该做为 `peerDependency` 而不是做为 `dependency` 使用）并且把这个报告给相关模块。

> 原文： *https://jkchao.github.io/typescript-book-chinese/error/common.html*

- 基于字符串的枚举
- 名义化类型
- 状态函数
- Bind 是有害的
- 柯里化
- 泛型的实例化类型
- 对象字面量的惰性初始化
- 类是有用的
- export default 被认为是有害的
- 减少 setter 属性的使用
- 创建数组
- 谨慎使用 —outFile
- TypeScript 中的静态构造函数
- 单例模式
- 函数参数
- Truthy
- 构建切换
- 类型安全的 Event Emitter
- Reflect Metadata
- 协变与逆变

基于字符串的枚举

有时你需要在公共的键下收集一些字符串的集合。在 `TypeScript 2.4` 以前，它仅支持基于数字类型的枚举，如果你在使用 `TypeScript 2.4` 以下的版本，你通过可以使用字符串字面量类型与联合类型组合使用创建基于字符串枚举类型的方式。

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/stringBasedEmuns.html*

名义化类型

TypeScript 的类型系统是结构化的，这里有个主要的原因。然而，在实际项目中可能是你想区分两个类型不同的变量，尽管它们具有相同的结构。一个非常常见的实际用例是身份的验证（它们可能只是在 C# 或者 Java 中表示一个它们语义化名字的字符串）。

这有一些社区推荐的方式，我按照个人爱好降序排列：

使用字面量类型

这种模式使用泛型和字面量类型：

```
// 泛型 Id 类型
type Id<T extends string> = {
  type: T;
  value: string;
};

// 特殊的 Id 类型
type FooId = Id<'foo'>;
type BarId = Id<'bar'>;

// 可选：构造函数
const createFoo = (value: string): FooId => ({ type: 'foo', value });
const createBar = (value: string): BarId => ({ type: 'bar', value });

let foo = createFoo('sample');
let bar = createBar('sample');

foo = bar; // Error
foo = foo; // Okey
```

- 优点
    - 不需要类型断言。
- 缺点
    - 如上结构 `{type,value}` 可能不那么尽如人意，而且需要服务器序列化支持。

使用枚举

TypeScript 中枚举 提供一定程度的名义化类型。如果两个枚举的命名不相同，则它们类型不相等。我们可以利用这个事实来为结构上兼容的类型，提供名义化类型。

解决办法包括：

- 创建一个只有名字的枚举；
- 利用这个枚举与实际结构体创建一个交叉类型（`&`）。
  如下所示，当实际结构体仅仅是一个字符串时：

```
// FOO
enum FooIdBrand {};
type FooId = FooIdBrand & string;

// BAR
enum BarIdBrand {};
type BarId = BarIdBrand & string;

// user

let fooId: FooId;
let barId: BarId;

// 类型安全
fooId = barId; // error
barId = fooId; // error

// 创建一个新的
fooId = 'foo' as FooId;
barId = 'bar' as BarId;

// 两种类型都与基础兼容
let str: string;
str = fooId;
str = barId;
```

使用接口

因为 `number` 类型与 `enum` 类型在类型上是兼容的，因此我们不能使用上述提到的方法来处理它们。取而代之，我们可以使用接口打破这种类型的兼容性。TypeScript 编译团队仍然在使用这种方法，因此它值得一提。使用 `_` 前缀和 `Brand` 后缀是一种我强烈推荐的惯例方法（TypeScript 也这么推荐）。

解决办法包括：

- 在类型上添加一个不用的属性，用来打破类型兼容性；
- 在需要的时候使用类型断言。

  如下所示：

```
// FOO
interface FooId extends String {
  _fooIdBrand: string; // 防止类型错误
}

// BAR
interface BarId extends String {
  _barIdBrand: string; // 防止类型错误
}

// 使用
let fooId: FooId;
let barId: BarId;

// 类型安全
fooId = barId; // error
barId = fooId; // error
fooId = <FooId>barId; // error
barId = <BarId>fooId; // error

// 创建新的
fooId = 'foo' as any;
barId = 'bar' as any;

// 如果你需要以字符串作为基础
var str: string;
str = fooId as any;
str = barId as any;
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/nominalTyping.html*

状态函数

其他编程语言有一个共同特征，它们使用 `static` 关键字来增加函数变量的生命周期（不是范围），使其超出函数的调用范围，如 C 语言中的实现：

```
void called () {
    static count = 0;
    count++;
    printf("Called : %d", count);
}

int main () {
    called(); // Called : 1
    called(); // Called : 2
    return 0;
}
```

由于 JavaScript（TypeScript）并没有静态函数的功能，你可以使用一个包裹着本地变量的抽象变量，如使用 `class`：

```
const { called } = new class {
  count = 0;
  called = () => {
    this.count++;
    console.log(`Called : ${this.count}`);
  };
}();

called(); // Called : 1
called(); // Called : 2
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/statefulFunctions.html*

Bind 是有害的

TIP

译者注：在这个 PR 下，已经解决 `bind`、`call`、`apply` 类型正确推导的问题，预计在 3.2 版本中发布。

这是在 `lib.d.ts` 中 `bind` 的定义：

```
bind(thisArg: any, ...argArray: any[]): any
```

你可以看到他的返回值是 `any`，这意味着在函数上调用 `bind` 会导致你在原始函数调用签名上将会完全失去类型的安全检查。

如下所示：

```
function twoParams(a: number, b: number) {
  return a + b;
}

let curryOne = twoParams.bind(null, 123);
curryOne(456); // ok
curryOne('456'); // ok
```

一个更好的方式的是使用类型注解的箭头函数：

```
function twoParams(a: number, b: number) {
  return a + b;
}

let curryOne = (x: number) => twoParams(123, x);
curryOne(456); // ok
curryOne('456'); // Error
```

如果你想用一个柯里化的函数，你可以看看此章节：

类成员

另一个常见用途是在传递类函数时使用 `bind` 来确保 `this` 的正确值，不要这么做。

在接下来的示例中，如果你使用了 `bind`，你将会失去函数参数的类型安全：

```
class Adder {
  constructor(public a: string) {}

  add(b: string): string {
    return this.a + b;
  }
}

function useAdd(add: (x: number) => number) {
  return add(456);
}

let adder = new Adder('mary had a little ?');
useAdd(adder.add.bind(adder)); // 没有编译的错误
useAdd(x => adder.add(x)); // Error: number 不能分配给 string
```

如果你想传递一个类成员的函数，使用箭头函数。例如：

```
class Adder {
  constructor(public a: string) {}

  // 此时，这个函数可以安全传递
  add = (b: string): string => {
    return this.a + b;
  };
}
```

另一种方法是手动指定要绑定的变量的类型：

```
const add: typeof adder.add = adder.add.bind(adder);
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/bind.html*

柯里化

仅仅需要使用一系列箭头函数:

```
// 一个柯里化函数
let add = (x: number) => (y: number) => x + y;

// 简单使用
add(123)(456);

// 部分应用
let add123 = add(123);

// fully apply the function
add123(456);
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/curry.html*

泛型的实例化类型

假如你有一个具有泛型参数的类型，如一个类 `Foo` ：

```
class Foo<T> {
  foo: T;
}
```

你想为一个特定的类型创建单独的版本，可以通过将它拷贝到一个新变量里，并且用具体类型代替泛型的类型注解的方式来实现。例如，如果你想有一个类： `Foo<number>` ：

```
class Foo<T> {
  foo: T;
}

const FooNumber = Foo as { new (): Foo<number> }; // ref 1
```

在 `ref 1` 中，你说 `FooNumber` 与 `Foo` 相同，但是，只是将其看作使用 `new` 运算符调用时的一个 `Foo<Number>` 实例。

继承

类型断言模式是不安全的，因为编译器相信你在做正确的事情。在其他语言中用于类的常见模式是使用继承：

```
class FooNumber extends Foo<number> {}
```

WARNING

这里需要注意的一点，如果你在基类上使用修饰器，继承类可能没有与基类相同的行为（它不再被修饰器包裹）。

当然，如果你不需要一个单独的类，你仍然写出一个有效的强制/断言模式，因此在开始时，我们便展示出了普通的断言模式：

```
function id<T>(x: T) {
  return x;
}

const idNum = id as { (x: number): number };
```

> 灵感来源于: *stackoverflow question*

> 原文*: https://jkchao.github.io/typescript-book-chinese/tips/typeInstantiation.html*

对象字面量的惰性初始化

在 JavaScript 中，像这样用字面量初始化对象的写法十分常见：

```
let foo = {};
foo.bar = 123;
foo.bas = 'Hello World';
```

但在 TypeScript 中，同样的写法就会报错：

```
let foo = {};
foo.bar = 123; // Error: Property 'bar' does not exist on type '{}'
foo.bas = 'Hello World'; // Error: Property 'bas' does not exist on type '{}'
```

这是因为 TypeScript 在解析 `let foo = {}` 这段赋值语句时，会进行“类型推断”：它会认为等号左边 `foo` 的类型即为等号右边 `{}` 的类型。由于 `{}` 本没有任何属性，因此，像上面那样给 `foo` 添加属性时就会报错。

最好的解决方案

最好的解决方案就是在为变量赋值的同时，添加属性及其对应的值：

```
let foo = {
  bar: 123,
  bas: 'Hello World'
};
```

这种写法也比较容易通过其他人或工具的代码审核，对后期维护也是有利的。

> 以下的快速解决方案采用惰性的思路，本质上是在初始化变量时忘了添加属性的做法。

快速解决方案

如果你的 JavaScript 项目很大，那么在迁移到 TypeScript 的时候，上面的做法可能会比较麻烦。此时，你可以利用 TypeScript 的“类型断言”机制让代码顺利通过编译：

```
let foo = {} as any;
foo.bar = 123;
foo.bas = 'Hello World';
```

折中的解决方案

当然，总是用 `any` 肯定是不好的，因为这样做其实是在想办法绕开 TypeScript 的类型检查。那么，折中的方案就是创建 `interface`，这样的好处在于：

- 方便撰写类型文档
- TypeScript 会参与类型检查，确保类型安全

  请看以下的示例：

```
interface Foo {
  bar: number;
  bas: string;
}

let foo = {} as Foo;
foo.bar = 123;
foo.bas = 'Hello World';
```

使用 `interface` 可以确保类型安全，比如这种情况：

```
interface Foo {
  bar: number;
  bas: string;
}

let foo = {} as Foo;
foo.bar = 123;
foo.bas = 'Hello World';

// 然后我们尝试这样做：
foo.bar = 'Hello Stranger'; // 错误：你可能把 `bas` 写成了 `bar`，不能为数字类型的属性赋值字符串
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/lazyObjectLiteralInitialization.html*

类是有用的

以下结构在应用中很常见:

```
function foo() {
  let someProperty;

  // 一些其他的初始化代码

  function someMethod() {
    // 用 someProperty 做一些事情
    // 可能有其他属性
  }

  // 可能有其他的方法
  return {
    someMethod
    // 可能有其他方法
  };
}
```

它被称为模块模式（利用 JavaScript 的闭包）。

如果你使用文件模块（你确实应该将全局变量视为错误），文件中的代码与示例一样，都不是全局变量。

然而，开发者有时会写以下类似代码:

```
let someProperty;

function foo() {
  // 一些初始化代码
}

foo();
someProperty = 123; // 其他初始化代码

// 一些其它未导出

// later
export function someMethod() {}
```

尽管我并不是一个特别喜欢使用继承的人，但是我确实发现让开发者使用类，可以在一定程度上更好的组织他们的代码，例如：

```
class Foo {
  public someProperty;

  constructor() {
    // 一些初始化内容
  }

  public someMethod() {
    // ..code
  }

  public someUtility() {
    // .. code
  }
}

export = new Foo();
```

这并不仅仅有利于开发者，在创建基于类的更出色可视化工具中，它更常见。并且，这有利于项目的理解和维护。

TIP

在浅层次的结构中，如果它们能够提供明显的重复使用和减少模版的好处，那么在这个观点里，我并没有错误。

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/classAreUseful.html*

export default 被认为是有害的

假如你有一个包含以下内容的 `foo.ts` 文件：

```
class Foo {}

export default Foo;
```

你可能会使用 ES6 语法导入它（在 `bar.ts` 里）：

```
import Foo from './foo';
```

这存在一些可维护性的问题：

- 如果你在 `foo.ts` 里重构 `Foo`，在 `bar.ts` 文件中，它将不会被重新命名；
- 如果你最终需要从 `foo.ts` 文件中导出更多有用的信息（在你的很多文件中都存在这种情景），那么你必须兼顾导入语法。
  由于这些原因，我推荐在导入时使用简单的 `export` 与解构的形式，如 `foo.ts`：

```
export class Foo {}
```

接着：

```
import { Foo } from './Foo';
```

下面，我将会介绍更多的原因。

可发现性差

默认导出的可发现性非常差，你不能智能的辨别一个模块它是否有默认导出。

在使用默认导出时，你什么也没有得到（可能它有默认导出，可能它没有）。

```
import /* here */ from 'something';
```

没有默认导出，你可以用以下方式获取智能提示：

```
import /* here */ 'something';
```

自动完成

不管你是否了解导出，你都可以在 `import { / here / } from './foo'` 的 `here` 位置，来了解导出模块的信息。

CommonJS 互用

对于必须使用 `const { default } = require('module/foo')` 而不是 `const { Foo } = require('module/foo')` 的 CommonJS 的用户来说，这会是一个糟糕的体验。当你导入一个模块时，你很可能想重命名 `default` 作为导入的名字。

防止拼写错误

当你在开发时使用 `import Foo from './foo'` 时，并不会得到有关于拼写的任何错误，其他人可能会这么写 `import foo from './foo'`；

再次导出

再次导出是没必要的，但是在 `npm` 包的根文件 `index` 却是很常见。如：`import Foo from './foo'；export { Foo }`（默认导出）VS `export * from './foo'` （命名导出）。

动态导入

在动态的 `import` 中，默认导出会以 `default` 的名字暴露自己，如：

```
const HighChart = await import('https://code.highcharts.com/js/es-modules/masters/highcharts.src.js');
Highcharts.default.chart('container', { ... }); // Notice `.default`
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/avoidExportDefault.html*

减少 setter 属性的使用

倾向于使用更精确的 `set/get` 函数（如 `setBar`, `getBar`），减少使用 `setter/getter`；

考虑以下代码：

```
foo.bar = {
  a: 123,
  b: 456
};
```

存在 `setter/getter` 时：

```
class Foo {
  a: number;
  b: number;
  set bar(value: { a: number; b: number }) {
    this.a = value.a;
    this.b = value.b;
  }
}

let foo = new Foo();
```

这并不是 `setter` 的一个好的使用场景，当开发人员阅读第一段代码时，不知道将要更改的所有内容的上下文。然而，当开发者使用 `foo.setBar(value)`，他可能会意识到在 `foo` 里可能会引起一些改变。

> 原文：*https://jkchao.github.io/typescript-book-chinese/tips/limitPropertySetters.html*

创建数组

创建数组十分简单：

```
const foo: string[] = [];
```

你也可以在创建数组时使用 ES6 的 `Array.prototype.fill` 方法为数组填充数据：

```
const foo: string[] = new Array(3).fill('');
console.log(foo); // 会输出 ['','','']
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/createArrays.html*

谨慎使用 —outFile

由于以下几点原因，你应该谨慎使用 `—outFile` 选项：

- 运行时的错误；
- 快速编译；
- 全局作用域；
- 难以分析；
- 难以扩展；
- `_references`；
- 代码重用；
- 多目标；
- 单独编译；

运行时的错误

如果你的代码依赖于上文中的 JavaScript，你可能会在运行时得到错误：

- 类的继承在运行时中断。
  有如下 `foo.ts`：

```
class Foo {}
```

以及 `bar.ts`：

```
class Bar extends Foo {}
```

如果你没有按正确的顺序编译它（例如：`tsc bar.ts foo.ts`），虽然它能够被编译成功，但是会在运行时抛出 `ReferenceError` 的错误。

- 模块拆分在运行时会失败。
  `foo.ts`：

```
namespace App {
  export const foo = 123;
}
```

`bar.ts`：

```
namespace App {
  export const bar = foo + 456;
}
```

与上文中一致，当你没有用正确的顺序编译它，它会在运行时将 `NaN` 赋值给 `bar`；

快速编译

如果你使用 `–out` 编译选项，而没有使用一些 `hacks` 时，单独的 `.ts` 文件是不会被编译成单独的 `.js` 文件。 `–out` 选项实际上使用了较慢的构建方式。

此外，由于 source map 基于长度编码，且对位置信息敏感，因此，大部分 source map 都会在编译时重新构建（如果你使用 source map）。

全局作用域

当然，你可以使用命名空间，但是它仍然在 `window` 上（如果你在浏览器中打开），命名空间仅仅是一个临时的解决方式。 `///<reference` 也不例外，它会引入一个难以维护的全局上下文。

如果你的公司有多个独立工作的团队，当有人决定尝试集成两个程序编写 app 时，则很可能存在命名冲突。

难以分析

我们希望提供更多代码分析工具。如果你提供调用链的依赖关系，这些将会变得简单。

难以扩展

实际上这是运行时的随机错误 + 编译时间时间慢 + 难以理解的代码的结果。

_references.ts

它并没有被 `tsconfig.json` 支持：Comment，你需要手动对文件排序。

代码重用

如果你想在另一个项目中重用存在隐式依赖关系的代码，如果没有错误提示，很难移植它。

多目标

如果你想在 nodejs 之类的环境下重用在浏览器中的代码（如：testing APIS），你将不得不将其转换到新的模块系统或者使用不好的 `hacks`，让 nodejs 的 `global` 成为你的新的全局变量（如：`window`）。

单独编译

文件无法被单独编译，如 `a.ts`：

```
namespace M {
  var s = t;
}
```

根据不同的 `b.ts` 的形式，它将有不同的输出：

```
namespace M {
  export var t = 5;
}
```

或者：

```
var t = 5;
```

因此 `a.ts` 不能被单独编译；

总结

`—out` 做的是一些构建工具的工作，这些构建工具也可以受益于外部模块所提供的依赖，因此如果你愿意，我们推荐你使用外部模块，让构建工具创建单文件的 `.js`。

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/outFileCaution.html*

TypeScript 中的静态构造函数

TypeScript 中的 `class` （JavaScript 中的 `class` ）没有静态构造函数的功能，但是你可以通过调用它自己来获取相同的效果：

```
class MyClass {
  static initalize() {
    //
  }
}

MyClass.initalize();
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/staticConstructors.html*

单例模式

传统的单例模式可以用来解决所有代码必须写到 `class` 中的问题：

```
class Singleton {
  private static instance: Singleton;
  private constructor() {
    // ..
  }

  public static getInstance() {
    if (!Singleton.instance) {
      Singleton.instance = new Singleton();
    }

    return Singleton.instance;
  }

  someMethod() {}
}

let someThing = new Singleton(); // Error: constructor of 'singleton' is private

let instacne = Singleton.getInstance(); // do some thing with the instance
```

然而，如果你不想延迟初始化，你可以使用 `namespace` 替代：

```
namespace Singleton {
  // .. 其他初始化的代码

  export function someMethod() {}
}

// 使用
Singleton.someMethod();
```

WARNING

单例只是全局的一个别称。

对大部分使用者来说，`namespace` 可以用模块来替代。

```
// someFile.ts
// ... any one time initialization goes here ...
export function someMethod() {}

// Usage
import { someMethod } from './someFile';
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/singletonPatern.html*

函数参数

如果你有一个含有很多参数或者相同类型参数的函数，那么你可能需要考虑使用对象的形式来替代这些函数参数：

如下一个函数：

```
function foo(flagA: boolean, flagB: boolean) {
  // 函数主体
}
```

像这样的函数，你可能会很容易错误的调用它，如 `foo(flagB, flagA)` ，并且你并不会从编译器得到想要的帮助。

你可以使用接收一个对象参数的形式：

```
function foo(
  config: {
    flagA: boolean;
    flagB: boolean;
  }
) {
  const { flagA, flagB } = config;
}
```

现在，函数将会被 `foo({ flagA, flagB })` 的形式调用，这样有利于发现错误及代码审查。

TIP

如果你的函数足够简单，并且你不希望增加代码，忽略这个建议。

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/functionParameters.html*

Truthy

JavaScript 有一个 `truthy` 概念，即在某些场景下会被推断为 `true`，例如除 `0` 以外的任何数字：

```
if (123) {
  // 将会被推断出 `true`
  console.log('Any number other than 0 is truthy');
}
```

你可以用下表来做参考：

Variable Type	When it is falsy	When it is truthy
boolean	false	true
string	' ' (empty string)	any other string
number	0 NaN	any other number
null	always	never
Any other Object including empty ones like {},[]	never	always

明确的

通过操作符 `!!`，你可以很容易的将某些值转化为布尔类型的值，例如：`!!foo`，它使用了两次 `!`，第一个 `!` 用来将其（在这里是 `foo`）转换为布尔值，但是这一操作取得的是其取反后的值，第二个取反时，能得到真正的布尔值。

这在很多地方都可以看到：

```
// Direct variables
const hasName = !!name;

// As members of objects
const someObj = {
  hasName: !!name
};

// ReactJS
{
```

```
    !!someName && <div>{someName}</div>;
}
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/truthy.html*

构建切换

通常可以根据 JavaScript 项目的运行环境来切换它们，通过 webpack 可以很轻松的切换环境变量。

在你的 `package.json script` 里，添加不同的编译目标：

```
"build:test": "webpack -p --config ./src/webpack.config.js",
"build:prod": "webpack -p --define process.env.NODE_ENV='\"production\"' --config ./src/webpack.config.js"
```

当然，假设你已经安装了 webpack `npm install webpack`，现在，你可以运行 `npm run build:test` 了。

使用环境变量也超级简单：

```
/**
 * This interface makes sure we don't miss adding a property to both `prod` and `test`
 */
interface Config {
  someItem: string;
}

/**
 * We only export a single thing. The config.
 */
export let config: Config;

/**
 * `process.env.NODE_ENV` definition is driven from webpack
 *
 * The whole `else` block will be removed in the emitted JavaScript
 *  for a production build
 */
if (process.env.NODE_ENV === 'production') {
  config = {
    someItem: 'prod'
  };
  console.log('Running in prod');
} else {
```

```
  config = {
    someItem: 'test'
  };
  console.log('Running in test');
}
```

TIP

我们使用 `process.env.NODE_ENV` 仅仅是因为绝大多数 JavaScript 库中都使用此变量，例如：`React` 。

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/buildToggles.html*

类型安全的 Event Emitter

通常来说，在 Node.js 与传统的 JavaScript 里，你有一个单一的 Event Emitter，你可以用它来为不同的事件添加监听器。

```
const emitter = new EventEmitter();

// Emit
emitter.emit('foo', foo);
emitter.emit('bar', bar);

// Listen
emitter.on('foo', foo => console.log(foo));
emitter.on('bar', foo => console.log(bar));
```

实际上，在 `EventEmitter` 内部以映射数组的形式存储数据：

```
{ foo: [fooListeners], bar: [barListeners] }
```

为了事件的类型安全，你可以为每个事件类型创建一个 emitter：

```
const onFoo = new TypedEvent<Foo>();
const onBar = new TypedEvent<Bar>();

// Emit:
onFoo.emit(foo);
onBar.emit(bar);

// Listen:
onFoo.on(foo => console.log(foo));
onBar.on(bar => console.log(bar));
```

它一些优点：

- 事件的类型，能以变量的形式被发现。
- Event Emitter 非常容易被重构。
- 事件数据结构是类型安全的。

参考 TypedEvent

```
export interface Listener<T> {
  (event: T): any;
}

export interface Disposable {
  dispose(): any;
}

export class TypedEvent<T> {
  private listeners: Listener<T>[] = [];
  private listenersOncer: Listener<T>[] = [];

  public on = (listener: Listener<T>): Disposable => {
    this.listeners.push(listener);

    return {
      dispose: () => this.off(listener)
    };
  };

  public once = (listener: Listener<T>): void => {
    this.listenersOncer.push(listener);
  };

  public off = (listener: Listener<T>) => {
    const callbackIndex = this.listeners.indexOf(listener);
    if (callbackIndex > -1) this.listeners.splice(callbackIndex, 1);
  };

  public emit = (event: T) => {
    this.listeners.forEach(listener => listener(event));

    this.listenersOncer.forEach(listener => listener(event));

    this.listenersOncer = [];
  };

  public pipe = (te: TypedEvent<T>): Disposable => {
    return this.on(e => te.emit(e));
  };
}
```

> **原文**: *https://jkchao.github.io/typescript-book-chinese/tips/typesafeEventEmitter.html*

Reflect Metadata

基础

Reflect Metadata 是 ES7 的一个提案，它主要用来在声明的时候添加和读取元数据。TypeScript 在 1.5+ 的版本已经支持它，你只需要：

- `npm i reflect-metadata –save`。
- 在 `tsconfig.json` 里配置 `emitDecoratorMetadata` 选项。
  Reflect Metadata 的 API 可以用于类或者类的属性上，如：

```
function metadata(
  metadataKey: any,
  metadataValue: any
): {
  (target: Function): void;
  (target: Object, propertyKey: string | symbol): void;
};
```

`Reflect.metadata` 当作 `Decorator` 使用，当修饰类时，在类上添加元数据，当修饰类属性时，在类原型的属性上添加元数据，如：

```
@Reflect.metadata('inClass', 'A')
class Test {
  @Reflect.metadata('inMethod', 'B')
  public hello(): string {
    return 'hello world';
  }
}

console.log(Reflect.getMetadata('inClass', Test)); // 'A'
console.log(Reflect.getMetadata('inMethod', new Test(), 'hello')); // 'B'
```

它具有诸多使用场景。

获取类型信息

譬如在 `vue-property-decorator` 6.1 及其以下版本中，通过使用 `Reflect.getMetadata` API，`Prop` Decorator 能获取属性类型传至 Vue，简要代码如下：

```
function Prop(): PropertyDecorator {
  return (target, key: string) => {
    const type = Reflect.getMetadata('design:type', target, key);
    console.log(`${key} type: ${type.name}`);
    // other...
  };
}

class SomeClass {
  @Prop()
  public Aprop!: string;
}
```

运行代码可在控制台看到 `Aprop type: string` 。除能获取属性类型外，通过 `Reflect.getMetadata("design:paramtypes", target, key)` 和 `Reflect.getMetadata("design:returntype", target, key)` 可以分别获取函数参数类型和返回值类型。

自定义 metadataKey

除能获取类型信息外，常用于自定义 `metadataKey` ，并在合适的时机获取它的值，示例如下：

```
function classDecorator(): ClassDecorator {
  return target => {
    // 在类上定义元数据，key 为 `classMetaData`，value 为 `a`
    Reflect.defineMetadata('classMetaData', 'a', target);
  };
}

function methodDecorator(): MethodDecorator {
  return (target, key, descriptor) => {
    // 在类的原型属性 'someMethod' 上定义元数据，key 为 `methodMetaData`，value 为 `b`
    Reflect.defineMetadata('methodMetaData', 'b', target, key);
  };
}

@classDecorator()
class SomeClass {
  @methodDecorator()
  someMethod() {}
```

```
}

Reflect.getMetadata('classMetaData', SomeClass); // 'a'
Reflect.getMetadata('methodMetaData', new SomeClass(), 'someMethod'); // 'b'
```

例子

控制反转和依赖注入

在 Angular 2+ 的版本中，控制反转与依赖注入便是基于此实现，现在，我们来实现一个简单版：

```
type Constructor<T = any> = new (...args: any[]) => T;

const Injectable = (): ClassDecorator => target => {};

class OtherService {
  a = 1;
}

@Injectable()
class TestService {
  constructor(public readonly otherService: OtherService) {}

  testMethod() {
    console.log(this.otherService.a);
  }
}

const Factory = <T>(target: Constructor<T>): T => {
  // 获取所有注入的服务
  const providers = Reflect.getMetadata('design:paramtypes', target); // [OtherService]
  const args = providers.map((provider: Constructor) => new provider());
  return new target(...args);
};

Factory(TestService).testMethod(); // 1
```

Controller 与 Get 的实现

如果你在使用 TypeScript 开发 Node 应用，相信你对 `Controller`、`Get`、`POST` 这些

Decorator，并不陌生：

```
@Controller('/test')
class SomeClass {
  @Get('/a')
  someGetMethod() {
    return 'hello world';
  }

  @Post('/b')
  somePostMethod() {}
}
```

这些 Decorator 也是基于 `Reflect Metadata` 实现，这次，我们将 `metadataKey` 定义在 `descriptor` 的 `value` 上：

```
const METHOD_METADATA = 'method';
const PATH_METADATA = 'path';

const Controller = (path: string): ClassDecorator => {
  return target => {
    Reflect.defineMetadata(PATH_METADATA, path, target);
  }
}

const createMappingDecorator = (method: string) => (path: string):
MethodDecorator => {
  return (target, key, descriptor) => {
    Reflect.defineMetadata(PATH_METADATA, path, descriptor.value);
    Reflect.defineMetadata(METHOD_METADATA, method, descriptor.value);
  }
}

const Get = createMappingDecorator('GET');
const Post = createMappingDecorator('POST');
```

接着，创建一个函数，映射出 `route`：

```
function mapRoute(instance: Object) {
  const prototype = Object.getPrototypeOf(instance);

  // 筛选出类的 methodName
```

```
  const methodsNames = Object.getOwnPropertyNames(prototype)
                                .filter(item => !isConstructor(item) &&
isFunction(prototype[item]));
  return methodsNames.map(methodName => {
    const fn = prototype[methodName];

    // 取出定义的 metadata
    const route = Reflect.getMetadata(PATH_METADATA, fn);
    const method = Reflect.getMetadata(METHOD_METADATA, fn);
    return {
      route,
      method,
      fn,
      methodName
    }
  })
};
```

因此，我们可以得到一些有用的信息：

```
Reflect.getMetadata(PATH_METADATA, SomeClass); // '/test'

mapRoute(new SomeClass());

/**
 * [{
 *    route: '/a',
 *    method: 'GET',
 *    fn: someGetMethod() { ... },
 *    methodName: 'someGetMethod'
 *  },{
 *    route: '/b',
 *    method: 'POST',
 *    fn: somePostMethod() { ... },
 *    methodName: 'somePostMethod'
 * }]
 *
 */
```

最后，只需把 `route` 相关信息绑在 `express` 或者 `koa` 上就 ok 了。

> 原文: *https://jkchao.github.io/typescript-book-chinese/tips/metadata.html*

协变与逆变

> 原文链接: *what are covariance and contravariance*

子类型 在编程理论上是一个复杂的话题，而他的复杂之处来自于一对经常会被混淆的现象，我们称之为协变与逆变。这篇文章将会解释上述两个概念。

开始文章之前我们先约定如下的标记:

- `A ≼ B` 意味着 `A` 是 `B` 的子类型。
- `A → B` 指的是以 `A` 为参数类型，以 `B` 为返回值类型的函数类型。
- `x : A` 意味着 `x` 的类型为 `A`。

一个有趣的问题

假设我有如下三种类型:

> *`Greyhound ≼ Dog ≼ Animal`*

`Greyhound` （灰狗）是 `Dog` （狗）的子类，而 `Dog` 则是 `Animal` （动物）的子类。由于子类型通常是可传递的，因此我们也称 `Greyhound` 是 `Animal` 的子类。

问题: 以下哪种类型是 `Dog → Dog` 的子类呢？

- `Greyhound → Greyhound`
- `Greyhound → Animal`
- `Animal → Animal`
- `Animal → Greyhound`
  让我们来思考一下如何解答这个问题。首先我们假设 `f` 是一个以 `Dog → Dog` 为参数的函数。它的返回值并不重要，为了具体描述问题，我们假设函数结构体是这样的: `f : (Dog → Dog) → String`。

现在我想给函数 `f` 传入某个函数 `g` 来调用。我们来瞧瞧当 `g` 为以上四种类型时，会发生什么情况。

1. 我们假设 `g : Greyhound → Greyhound`，`f(g)` 的类型是否安全？

不安全。因为参数 `(g)` 有可能是一个不同于灰狗但又是狗的子类，例如 `GermanShepherd` （牧羊犬）。

2. 我们假设 `g : Greyhound → Animal`，`f(g)` 的类型是否安全？

不安全。理由同(1)。

3. 我们假设 `g : Animal → Animal` ， `f(g)` 的类型是否安全？

不安全。因为 `f` 有可能在调用完参数之后，让返回值，也就是 `Animal` （动物）狗叫。并非所有动物都会狗叫。

4. 我们假设 `g : Animal → Greyhound` ， `f(g)` 的类型是否安全？

是的，它的类型是安全的。首先， `f` 可能会以任何狗的品种来作为参数调用，而所有的狗都是动物。其次，它可能会假设结果是一条狗，而所有的灰狗都是狗。

展开讲讲？

如上所述，我们得出结论：

> `(Animal → Greyhound) ≼ (Dog → Dog)`

返回值类型很容易理论：灰狗是狗的子类。但参数类型则是相反的：动物是狗的父类！

用合适的术语来描述这个奇怪的表现，可以说我们允许一个函数类型中，返回值类型是协变的，而参数类型是逆变的。返回值类型是协变的，意思是 `A ≼ B` 就意味着 `(T → A) ≼ (T → B)` 。参数类型是逆变的，意思是 `A ≼ B` 就意味着 `(B → T) ≼ (A → T)` （ `A` 和 `B` 的位置颠倒过来了）。

一个有趣的现象：在 `TypeScript` 中， 参数类型是双向协变的，也就是说既是协变又是逆变的，而这并不安全。但是现在你可以在 `TypeScript 2.6` 版本中通过 `—strictFunctionTypes` 或 `—strict` 标记来修复这个问题。

那其他类型呢？

问题： `List<Dog>` 能否为 `List<Animal>` 的子类？

答案有点微妙。如果列表是不可变的（immutable），那么答案是肯定的，因为类型很安全。但是假如列表是可变的，那么答案绝对是否定的！

原因是，假设我需要一串 `List<Animal>` 而你传给我一串 `List<Dog>` 。由于我认为我拥有的是一串 `List<Animal>` ，我可能会尝试往列表插入一只 `Cat` 。那么你的 `List<Dog>` 里面就会有一只猫！类型系统不应该允许这种情况发生。

总结一下，我们可以允许不变的列表（immutable）在它的参数类型上是协变的，但是对于可变的列表（mutable），其参数类型则必须是不变的（invariant），既不是协变也不是逆变。

一个有趣的现象：在 `Java` 中，数组既是可变的，又是协变的。当然，这并不安全。

> 原文: *https://jkchao.github.io/typescript-book-*

> *chinese/tips/covarianceAndContravariance.html*

- 概览
- 程序
- 抽象语法树
- 扫描器
- 解析器
- 绑定器
- 检查器
- 发射器

概览

TypeScript 编译器源文件位于 `src/compiler` 目录下

它分为以下几个关键部分:

- Scanner 扫描器（`scanner.ts`）
- Parser 解析器（`parser.ts`）
- Binder 绑定器（`binder.ts`）
- Checker 检查器（`checker.ts`）
- Emitter 发射器（`emitter.ts`）
  每个部分在源文件中均有独立文件，本章稍后会对这些部分做解释。

BYOTS

我们有个名为 Bring Your Own TypeScript (BYOTS) 的项目，通过暴露内部接口让编译器 API 使用起来更简单。你可以在全局范围上暴露你 TypeScript 应用的本地变量。

语法和语义

语法正确并不意味着语义上也正确。下面的 TypeScript 代码，语法合法，但是语义却不正确

```
var foo: number = 'not a number';
```

`语义` 从自然语言角度意味着有意义，理解这个概念对你很有用。

处理概览

以下演示简单说明 TypeScript 编译器如何将上述几个关键部分组合在一起:

```
SourceCode（源码） ~~ 扫描器 ~~> Token 流
```

```
Token 流 ~~ 解析器 ~~> AST（抽象语法树）
```

```
AST ~~ 绑定器 ~~> Symbols（符号）
```

符号（`Symbol`）是 TypeScript 语义系统的主要构造块。如上所示，符号是绑定的结果。符号将 AST 中的声明节点与相同实体的其他声明相连。

符号和 AST 是检查器用来验证源代码语义的

```
1. AST + 符号 ~~ 检查器 ~~> 类型验证
```

最后，需要输出 JavaScript 时：

```
1. AST + 检查器 ~~ 发射器 ~~> JavaScript 代码
```

TypeScript 编译器中还有一些其他文件，为我们接下来介绍的很多关键部分提供实用工具。

文件：Utilities

`core.ts` ：TypeScript 编译器使用的核心工具集，重要的有：

- `let objectAllocator: ObjectAllocator` 是一个定义为全局单例的变量。提供以下定义：
  - `getNodeConstructor`（节点会在解析器 / AST 中介绍）
  - `getSymbolConstructor`（符号会在绑定器中介绍）
  - `getTypeConstructor`（类型会在检查器中介绍）
  - `getSignatureConstructor`（签名是索引，调用和构造签名）

文件：关键数据结构

`types.ts` 包含整个编译器中使用的关键数据结构和接口，这里列出一些关键部分：

- `SyntaxKind`AST 节点类型通过 `SyntaxKind` 枚举进行识别
- `TypeChecker` 类型检查器提供此接口
- `CompilerHost` 用于程序（`Program`）和系统之间的交互
- `Node` AST 节点

文件：系统

`system.ts`，TypeScript 编译器与操作系统的所有交互均通过 `System` 接口进行。接口及其实现（`WScript` 和 `Node`） 均定义在 `system.ts` 中。你可以将其视为操作环境（*OE, Operating Environment*）。

现在对主要文件有一个整体了解了，我们继续介绍程序（`Program`）的概念

> 原文：*https://jkchao.github.io/typescript-book-chinese/compiler/overview.html*

程序

程序定义在 `program.ts` 中。编译上下文在 TypeScript 编译器中被视为一个 `Program`，它包含 `SourceFile` 和编译选项

CompilerHost 的使用

CompilerHost 是与操作环境（OE, Operating Enviornment）进行交互的机制：

`Program` -使用-> `CompilerHost` -使用-> `System`

用 `CompilerHost` 作中间层的原因是可以让接口对 `Program` 的需求进行细粒度的调整，而无需考虑操作环境的需求。（例如：`Program` 无需关心 `System` 的 `fileExists` 函数）

对 `System` 而言还有其他的使用者（比如测试）

SourceFile

程序有个 API，用于获取 SourceFile：`getSourceFiles(): SourceFile[];`。得到的每个元素均是一棵抽象语法树的根节点（称做 `SourceFile`）

> 原文: *https://jkchao.github.io/typescript-book-chinese/compiler/program.html*

抽象语法树

Node 节点

节点是抽象语法树（AST） 的基本构造块。语法上，通常 `Node` 表示非末端（non-terminals）节点。但是，有些末端节点，如：标识符和文字也会保留在树中。

AST 节点文档由两个关键部分构成。一是节点的 `SyntaxKind` 枚举，用于标识 AST 中的类型。二是其接口，即实例化 AST 时节点提供的 API。

这里是 `interface Node` 的一些关键成员：

- `TextRange` 标识该节点在源文件中的起止位置。
- `parent?: Node` 当前节点（在 AST 中）的父节点
  `Node` 还有一些其他的成员，标志（flags）和修饰符（modifiers）等。你可以在源码中搜索 `interface Node` 来查看，而上面提到对节点的遍历是非常重要的。

SourceFile

- `SyntaxKind.SourceFile`
- `interface SourceFile`.
  每个 `SourceFile` 都是一棵 AST 的顶级节点，它们包含在 `Program` 中。

AST 技巧：访问子节点

有个工具函数 `ts.forEachChild` ，可以用来访问 AST 任一节点的所有子节点。

下面是简化的代码片段，用于演示如何工作：

```
export function forEachChild<T>(node: Node, cbNode: (node: Node) => T, cbNodeArray?: (nodes: Node[]) => T): T {
    if (!node) {
        return;
    }
    switch (node.kind) {
        case SyntaxKind.BinaryExpression:
            return visitNode(cbNode, (<BinaryExpression>node).left) ||
                visitNode(cbNode, (<BinaryExpression>node).operatorToken) ||
                visitNode(cbNode, (<BinaryExpression>node).right);
        case SyntaxKind.IfStatement:
            return visitNode(cbNode, (<IfStatement>node).expression) ||
```

```
                visitNode(cbNode, (<IfStatement>node).thenStatement) ||
                visitNode(cbNode, (<IfStatement>node).elseStatement);

        // .... 更多
```

该函数主要检查 `node.kind` 并据此判断 node 的接口，然后在其子节点上调用 `cbNode` 。但是，要注意该函数不会为所有子节点调用 `visitNode` （例如：SyntaxKind.SemicolonToken）。想获得某 AST 节点的所有子节点，只要调用该节点的成员函数 `.getChildren` 。

如下函数会打印 AST 节点详细信息：

```
function printAllChildren(node: ts.Node, depth = 0) {
  console.log(new Array(depth + 1).join('----'), ts.syntaxKindToName(node.kind), node.pos, node.end);
  depth++;
  node.getChildren().forEach(c => printAllChildren(c, depth));
}
```

我们进一步讨论解析器时会看到该函数的使用示例。

AST 技巧：SyntaxKind 枚举

`SyntaxKind` 被定义为一个常量枚举，如下所示：

```
export const enum SyntaxKind {
    Unknown,
    EndOfFileToken,
    SingleLineCommentTrivia,
    // ... 更多
```

这是个常量枚举，方便内联（例如： `ts.SyntaxKind.EndOfFileToken` 会变为 `1` ），这样在使用 AST 时就不会有处理引用的额外开销。但编译时需要使用 –preserveConstEnums 编译标志，以便枚举在运行时仍可用。JavaScript 中你也可以根据需要使用 `ts.SyntaxKind.EndOfFileToken` 。另外，可以用以下函数，将枚举成员转化为可读的字符串：

```
export function syntaxKindToName(kind: ts.SyntaxKind) {
  return (<any>ts).SyntaxKind[kind];
}
```

AST 杂项

杂项（Trivia）是指源文本中对正常理解代码不太重要的部分，例如：空白，注释，冲突标记。（为了保持轻量）杂项不会存储在 AST 中。但是可以视需要使用一些 `ts.*` API 来获取。

展示这些 API 前，你需要理解以下内容：

杂项的所有权

通常：

- token 拥有它后面 同一行 到下一个 token 之前的所有杂项
- 该行之后的注释都与下个的 token 相关
  对于文件中的前导（leading）和结束（ending）注释：
- 源文件中的第一个 token 拥有所有开始的杂项
- 而文件最后的一些列杂项则附加到文件结束符上，该 token 长度为 0

杂项 API

注释在多数基本使用中，都是让人关注的杂项。节点的注释可以通过以下函数获取：

函数	描述
`ts.getLeadingCommentRanges`	给定源文本及其位置，返回给定位置后第一个换行符到 token 本身之间的注释范围（可能需要结合 `ts.Node.getFullStart` 使用）。
`ts.getTrailingCommentRanges`	给定源文本及其位置，返回给定位置后第一个换行符之前的注释范围（可能需要结合 `ts.Node.getEnd` 使用）。

假设下面是某个源文件的一部分：

```
debugger;/*hello*/
    //bye
  /*hi*/    function
```

对 `function` 而言，`getLeadingCommentRanges` 仅返回最后的两个注释 `//bye` 和 `/hi/`。另外，而在 `debugger` 语句结束位置调用 `getTrailingCommentRanges` 会得到注释 `/hello/`。

Token Start 和 Full Start 位置

节点有所谓的 "token start" 和 "full start" 位置。

- Token Start：比较自然的版本，即文件中一个 token 的文本开始的位置。
- Full Start：是指扫描器从上一个重要 token 开始扫描的位置。
  AST 节点有 `getStart` 和 `getFullStart` API 用于获取以上两种位置，还是这个例子：

```
debugger;/*hello*/
    //bye
  /*hi*/    function
```

对 `function` 而言，token start 即 `function` 的位置，而 *full* start 是 `/hello/` 的位置。要注意，full start 甚至会包含前一节点拥有的杂项。

> 原文: *https://jkchao.github.io/typescript-book-chinese/compiler/ast.html*

扫描器

TypeScript 扫描器的源码均位于 `scanner.ts` 。在内部，由解析器控制扫描器将源码转化为抽象语法树（AST）。期望结果如下：

```
SourceCode ~~ 扫描器 ~~> Token 流 ~~ 解析器 ~~> AST
```

解析器对扫描器的使用

为避免重复创建扫描器造成的开销， `parser.ts` 中创建了一个扫描器的单例。解析器根据需要使用 `initializeState` 函数准备该扫描器。

下面是解析器中的实际代码的简化版，你可以运行它演示以上概念

`code/compiler/scanner/runScanner.ts`

```
import * as ts from 'ntypescript';

// 单例扫描器
const scanner = ts.createScanner(ts.ScriptTarget.Latest, /* 忽略杂项 */ true);

// 此函数与初始化使用的 `initializeState` 函数相似
function initializeState(text: string) {
  scanner.setText(text);
  scanner.setOnError((message: ts.DiagnosticMessage, length: number) => {
    console.error(message);
  });
  scanner.setScriptTarget(ts.ScriptTarget.ES5);
  scanner.setLanguageVariant(ts.LanguageVariant.Standard);
}

// 使用示例
initializeState(
  `
var foo = 123;
`.trim()
);

// 开始扫描
var token = scanner.scan();
```

```
while (token != ts.SyntaxKind.EndOfFileToken) {
  console.log(ts.formatSyntaxKind(token));
  token = scanner.scan();
}
```

该段代码输出以下内容:

```
VarKeyword
Identifier
FirstAssignment
FirstLiteralToken
SemicolonToken
```

扫描器状态

调用 `scan` 后，扫描器更新其局部状态（扫描位置，当前 token 详情等）。扫描器提供了一组工具函数获取当前扫描器状态。下例中，我们创建一个扫描器并用它识别 token 以及 token 在代码中的位置。

`code/compiler/scanner/runScannerWithPosition.ts`

```
// 使用示例
initializeState(
  `
var foo = 123;
`.trim()
);

// 开始扫描
var token = scanner.scan();
while (token != ts.SyntaxKind.EndOfFileToken) {
  let currentToken = ts.formatSyntaxKind(token);
  let tokenStart = scanner.getStartPos();
  token = scanner.scan();
  let tokenEnd = scanner.getStartPos();
  console.log(currentToken, tokenStart, tokenEnd);
}
```

该代码输出以下内容:

```
VarKeyword 0 3
```

```
Identifier 3 7
FirstAssignment 7 9
FirstLiteralToken 9 13
SemicolonToken 13 14
```

独立扫描器

即便 TypeScript 解析器有单例扫描器，你仍可以使用 `createScanner` 创建独立的扫描器，然后可以用 `setText` / `setTextPos` 随意扫描文件的不同位置。

> 原文: *https://jkchao.github.io/typescript-book-chinese/compiler/scanner.html*

解析器

TypeScript 解析器代码均位于 `praser.ts` 中。在内部，由解析器控制扫描器将源码转化为 AST。下面是期望结果的回顾：

```
源码 ~~ 扫描器 ~~> Token 流 ~~ 解析器 ~~> AST
```

解析器实现为一个单例（其原因类似扫描器，如果能重新初始化就不重新构建）。实际实现成 `namespace Parser`，包含解析器的各种状态变量和单例扫描器（`const scanner`）。该扫描器由解析器函数管理。

程序对解析器的使用

解析器由程序间接驱动（通过之前提到过的 `CompilerHost`）。基本上，简化的调用栈如下所示：

```
程序 ->
    CompilerHost.getSourceFile ->
        (全局函数 parser.ts).createSourceFile ->
            Parser.parseSourceFile
```

`parseSourceFile` 不仅准备好解析器的状态，还调用 `initializeState` 准备好扫描器的状态。然后使用 `parseSourceFileWorker` 继续解析源代码。

使用示例

深入解析器的内部之前，这里有个使用 TypeScript 解析器的示例，（使用 `ts.createSourceFile`）获取一个源文件的 AST 并打印它。

`code/compiler/parser/runParser.ts`

```
import * as ts from 'ntypescript';

function printAllChildren(node: ts.Node, depth = 0) {
  console.log(new Array(depth + 1).join('----'), ts.formatSyntaxKind(node.kind), node.pos, node.end);
  depth++;
  node.getChildren().forEach(c => printAllChildren(c, depth));
}

var sourceCode = `
```

```
var foo = 123;
`.trim();

var sourceFile = ts.createSourceFile('foo.ts', sourceCode, ts.ScriptTarget.ES5, true);
printAllChildren(sourceFile);
```

该段代码会打印以下内容：

```
SourceFile 0 14
---- SyntaxList 0 14
-------- VariableStatement 0 14
------------ VariableDeclarationList 0 13
---------------- VarKeyword 0 3
---------------- SyntaxList 3 13
-------------------- VariableDeclaration 3 13
------------------------ Identifier 3 7
------------------------ FirstAssignment 7 9
------------------------ FirstLiteralToken 9 13
------------ SemicolonToken 13 14
---- EndOfFileToken 14 14
```

如果把头向左倾，这个看起来像棵（右侧）树

解析器函数

如前所述， `parseSourceFile` 设置初始状态并将工作交给 `parseSourceFileWorker` 函数。

parseSourceFileWorker

该函数先创建一个 `SourceFile` AST 节点，然后从 `parseStatements` 函数开始解析源代码。一旦返回结果，就用额外信息（例如 `nodeCount` ， `identifierCount` 等） 完善 `SourceFile` 节点。

parseStatements

是最重要的 `parseXXX` 系函数之一（概念接下来介绍）。它根据扫描器返回的当前 token 来切换（调用相应的 `parseXXX` 函数），例如：如果当前 token 是一个 `SemicolonToken` （分号标记），就会调用 `paserEmptyStatement` 为空语句创建一个 AST 节点。

节点创建

解析器有一系列 `parseXXX` 函数用来创建相应类型为 `XXX` 的节点，通常在相应类型的节点出现时被（其他解析器函数）调用。该过程的典型示例是解析空语句（例如 `;;;;;` ）时要用的 `parseEmptyStatement()` 函数。下面是其全部代码：

```
function parseEmptyStatement(): Statement {
  let node = <Statement>createNode(SyntaxKind.EmptyStatement);
  parseExpected(SyntaxKind.SemicolonToken);
  return finishNode(node);
}
```

它展示了 3 个关键函数 `createNode` ， `parseExpected` 和 `finishNode` .

createNode

解析器函数 `function createNode(kind: SyntaxKind, pos?: number): Node` 负责创建节点，设置节点的 `SyntaxKind` （语法类别），和初始位置（默认使用当前扫描器状态提供的位置信息）。

parseExpected

解析器的 `parseExpected` 函数 `function parseExpected(kind: SyntaxKind, diagnosticMessage?: DiagnosticMessage): boolean` 会检查解析器状态中的当前 token 是否与指定的 `SyntaxKind` 匹配。如果不匹配，会报告传入的 `diagnosticMessage` （诊断消息），未传入则使用某种通用形式 `xxx expected` 进行报告。该函数内部用 `parseErrorAtPosition` 函数（使用扫描位置）提供良好的错误报告。

finishNode

解析器的 `finishNode` 函数 `function finishNode<T extends Node>(node: T, end?: number): T` 设置节点的 `end` 位置，并添加一些有用的信息，例如上下文标志（ `parserContextFlags` ）以及解析该节点前出现的错误（有错的话，就不能在增量解析中重用此 AST 节点）。

> 原文: *https://jkchao.github.io/typescript-book-chinese/compiler/parser.html*

绑定器

大多数的 JavaScript 转译器（transpiler）都比 TypeScript 简单，因为它们几乎没提供代码分析的方法。典型的 JavaScript 转换器只有以下流程：

```
源码 ~~扫描器~~> Tokens ~~解析器~~> AST ~~发射器~~> JavaScript
```

上述架构确实对于简化 TypeScript 生成 JavaScript 的理解有帮助，但缺失了一个关键功能，即 TypeScript 的语义系统。为了协助（检查器执行）类型检查，绑定器将源码的各部分连接成一个相关的类型系统，供检查器使用。绑定器的主要职责是创建符号（Symbols）。

符号

符号将 AST 中的声明节点与其它声明连接到相同的实体上。符号是语义系统的基本构造块。符号的构造器定义在 `core.ts`（绑定器实际上通过 `objectAllocator.getSymbolConstructor` 来获取构造器）。下面是符号构造器：

```
function Symbol(flags: SymbolFlags, name: string) {
  this.flags = flags;
  this.name = name;
  this.declarations = undefined;
}
```

`SymbolFlags` 符号标志是个标志枚举，用于识别额外的符号类别（例如：变量作用域标志 `FunctionScopedVariable` 或 `BlockScopedVariable` 等）

检查器对绑定器的使用

实际上，绑定器被检查器在内部调用，而检查器又被程序调用。简化的调用栈如下所示：

```
program.getTypeChecker ->
    ts.createTypeChecker（检查器中）->
        initializeTypeChecker（检查器中） ->
            for each SourceFile `ts.bindSourceFile`（绑定器中）
            // followed by
            for each SourceFile `ts.mergeSymbolTable`（检查器中）
```

SourceFile 是绑定器的工作单元，`binder.ts` 由 `checker.ts` 驱动。

绑定器函数

`bindSourceFile` 和 `mergeSymbolTable` 是两个关键的绑定器函数，我们来看下：

bindSourceFile

该函数主要是检查 `file.locals` 是否定义，如果没有则交给（本地函数） `bind` 来处理。

注意： `locals` 定义在节点上，其类型为 `SymbolTable` 。 `SourceFile` 也是一个节点（事实上是 AST 中的根节点）。

提示：TypeScript 编译器大量使用本地函数。本地函数很可能使用来自父函数的变量（通过闭包捕获）。例如 `bind` 是 `bindSourceFile` 中的一个本地函数，它或它调用的函数会设置 `symbolCount` 和 `classifiableNames` 等状态，然后将其存在返回的 `SourceFile` 中

bind

bind 能处理任一节点（不只是 `SourceFile` ），它做的第一件事是分配 `node.parent` （如果 `parent` 变量已设置，绑定器在 `bindChildren` 函数的处理中仍会再次设置）， 然后交给 `bindWorker` 做很多重活。最后调用 `bindChildren` （该函数简单地将绑定器的状态（如： `parent` ）存入函数本地变量中，接着在每个子节点上调用 `bind` ，然后再将状态转存回绑定器中）。现在我们看下 `bindWorker` ，一个更有趣的函数。

bindWorker

该函数依据 `node.kind` （ `SyntaxKind` 类型）进行切换，并将工作委托给合适的 `bindXXX` 函数（也定义在 `binder.ts` 中）。例如：如果该节点是 `SourceFile` 则（最终且仅当节点是外部文件模块时）调用 `bindAnonymousDeclaration`

bindXXX 函数

`bindXXX` 系函数有一些通用的模式和工具函数。其中最常用的一个是 `createSymbol` 函数，全部代码展示如下：

```
function createSymbol(flags: SymbolFlags, name: string): Symbol {
  symbolCount++;
  return new Symbol(flags, name);
}
```

如您所见，它简单地更新 `symbolCount` （一个 `bindSourceFile` 的本地变量），并使用指定的参数创建符号。

绑定器声明

符号与声明

节点和符号间的链接由几个函数执行。其中一个用于绑定 `SourceFile` 节点到源文件符号（外部模块的情况下）的函数是 `addDeclarationToSymbol`

注意：外部模块源文件的符号设置为 `flags : SymbolFlags.ValueModule` 和 `name: '"' + removeFileExtension(file.fileName) + '"'`.

```
function addDeclarationToSymbol(symbol: Symbol, node: Declaration, symbolFlags: SymbolFlags) {
  symbol.flags |= symbolFlags;

  // 创建 AST 节点到 symbol 的连接
  node.symbol = symbol;

  if (!symbol.declarations) {
    symbol.declarations = [];
  }
  // 将该节点添加为该符号的一个声明
  symbol.declarations.push(node);

  if (symbolFlags & SymbolFlags.HasExports && !symbol.exports) {
    symbol.exports = {};
  }

  if (symbolFlags & SymbolFlags.HasMembers && !symbol.members) {
    symbol.members = {};
  }

  if (symbolFlags & SymbolFlags.Value && !symbol.valueDeclaration) {
    symbol.valueDeclaration = node;
  }
}
```

重要的部分（译注：相关注释已添加到上面的代码中）：

- 创建一个从 AST 节点到符号的链接（`node.symbol`）
- 将节点添加为该符号的一个声明

声明

声明就是一个有可选的名字的节点。下面是 `types.ts` 中的定义:

```
interface Declaration extends Node {
  _declarationBrand: any;
  name?: DeclarationName;
}
```

绑定器容器

AST 的节点可以作容器。这决定了的节点及相关符号的 `SymbolTables` 的类别。容器是个抽象概念（没有相关的数据结构）。该概念由一些东西决定, `ContainerFlags` 枚举是其中之一。函数 `getContainerFlags` （位于 `binder.ts` ） 驱动此标志，如下所示:

```
function getContainerFlags(node: Node): ContainerFlags {
  switch (node.kind) {
    case SyntaxKind.ClassExpression:
    case SyntaxKind.ClassDeclaration:
    case SyntaxKind.InterfaceDeclaration:
    case SyntaxKind.EnumDeclaration:
    case SyntaxKind.TypeLiteral:
    case SyntaxKind.ObjectLiteralExpression:
      return ContainerFlags.IsContainer;

    case SyntaxKind.CallSignature:
    case SyntaxKind.ConstructSignature:
    case SyntaxKind.IndexSignature:
    case SyntaxKind.MethodDeclaration:
    case SyntaxKind.MethodSignature:
    case SyntaxKind.FunctionDeclaration:
    case SyntaxKind.Constructor:
    case SyntaxKind.GetAccessor:
    case SyntaxKind.SetAccessor:
    case SyntaxKind.FunctionType:
    case SyntaxKind.ConstructorType:
    case SyntaxKind.FunctionExpression:
    case SyntaxKind.ArrowFunction:
    case SyntaxKind.ModuleDeclaration:
    case SyntaxKind.SourceFile:
    case SyntaxKind.TypeAliasDeclaration:
      return ContainerFlags.IsContainerWithLocals;

```

```
    case SyntaxKind.CatchClause:
    case SyntaxKind.ForStatement:
    case SyntaxKind.ForInStatement:
    case SyntaxKind.ForOfStatement:
    case SyntaxKind.CaseBlock:
      return ContainerFlags.IsBlockScopedContainer;

    case SyntaxKind.Block:
      // 不要将函数内部的块直接当做块作用域的容器。
      // 本块中的本地变量应当置于函数中，否则下例中的 'x' 不会重新声明为一个块作用域的本地变量：
      //
      //     function foo() {
      //         var x;
      //         let x;
      //     }
      //
      // 如果将 'var x' 留在函数中，而将 'let x' 放到本块中（函数外），就不会有冲突了。
      //
      // 如果不在这里创建一个新的块作用域容器，'var x' 和 'let x' 都会进入函数容器本地中，这样就会有碰撞冲突。
      return isFunctionLike(node.parent) ? ContainerFlags.None : ContainerFlags.IsBlockScopedContainer;
  }

  return ContainerFlags.None;
}
```

该函数只在绑定器函数 `bindChildren` 中调用，会根据 `getContainerFlags` 的运行结果将节点设为 `container` 和（或） `blockScopedContainer` 。函数 `bindChildren` 如下所示：

```
// 所有容器节点都以声明顺序保存在一个链表中。
// 类型检查器中的 getLocalNameOfContainer 函数会使用该链表对容器使用的本地名称的唯一性做验证。
function bindChildren(node: Node) {
  // 在递归到子节点之前，我们先要保存父节点，容器和块容器。处理完弹出的子节点后，再将这些值存回原处。
  let saveParent = parent;
  let saveContainer = container;
  let savedBlockScopeContainer = blockScopeContainer;

  // 现在要将这个节点设为父节点，我们要递归它的子节点。
```

```
  parent = node;

  // 根据节点的类型，需要对当前容器或块容器进行调整。 如果当前节点是个容器，则自动将其视为当前的块容器。
  // 由于我们知道容器可能包含本地变量，因此提前初始化 .locals 字段。
  // 这样做是因为很可能需要将一些子（节点）置入 .locals 中（例如：函数参数或变量声明）。
  //
  // 但是，我们不会主动为块容器创建 .locals，因为通常块容器中不会有块作用域变量。
  // 我们不想为遇到的每个块都分配一个对象，大多数情况没有必要。
  //
  // 最后，如果是个块容器，我们就清理该容器中可能存在的 .locals 对象。这种情况常在增量编译场景中发生。
  // 由于我们可以重用上次编译的节点，而该节点可能已经创建了 locals 对象。
  // 因此必须清理，以免意外地从上次的编译中移动了过时的数据。
  let containerFlags = getContainerFlags(node);
  if (containerFlags & ContainerFlags.IsContainer) {
    container = blockScopeContainer = node;

    if (containerFlags & ContainerFlags.HasLocals) {
      container.locals = {};
    }

    addToContainerChain(container);
  } else if (containerFlags & ContainerFlags.IsBlockScopedContainer) {
    blockScopeContainer = node;
    blockScopeContainer.locals = undefined;
  }

  forEachChild(node, bind);

  container = saveContainer;
  parent = saveParent;
  blockScopeContainer = savedBlockScopeContainer;
}
```

您可能还记得绑定器函数中的这部分：`bindChildren` 由 `bind` 函数调用。我们得到这样的递归绑定：`bind` 调用 `bindChildren`，而 `bindChildren` 又为其每个子节点调用 `bind`

绑定器符号表

符号表（SymbolTable）实现为一个简单的 HashMap，下面是其接口（`types.ts`）：

```
interface SymbolTable {
  [index: string]: Symbol;
}
```

符号表通过绑定进行初始化，这里是编译器使用的一些符号表：

节点上：

```
locals?: SymbolTable;                    // 节点相关的本地变量
```

符号上：

```
members?: SymbolTable;                   // 类，接口或字面量实例成员
exports?: SymbolTable;                   // 模块导出
```

请注意： `bindChildren` 基于 `ContainerFlags` 初始化 `locals` （为 `{}` ）

符号表填充

符号表使用符号来填充，主要是通过调用 `declareSymbol` 来进行，如下所示的是该函数的全部代码：

```
/**
 * 为指定的节点声明一个符号并加入 symbols。标识名冲突时报告错误。
 * @param symbolTable - 要将节点加入进的符号表
 * @param parent - 指定节点的父节点的声明
 * @param node - 要添加到符号表的（节点）声明
 * @param includes - SymbolFlags，指定节点额外的声明类型（例如：export, ambient 等）
 * @param excludes - 不能在符号表中声明的标志，用于报告禁止的声明
 */
function declareSymbol(
  symbolTable: SymbolTable,
  parent: Symbol,
  node: Declaration,
  includes: SymbolFlags,
  excludes: SymbolFlags
): Symbol {
  Debug.assert(!hasDynamicName(node));

  // 默认导出的函数节点或类节点的符号总是"default"
  let name = node.flags & NodeFlags.Default && parent ? 'default' :
getDeclarationName(node);
```

```
  let symbol: Symbol;
  if (name !== undefined) {
    // 检查符号表中是否已有同名的符号。若没有，创建此名称的新符号并加入表中。
    // 注意，我们尚未给新符号指定任何标志。这可以确保不会和传入的 excludes 标志起冲突。
    //
    // 如果已存在的一个符号，查看是否与要创建的新符号冲突。
    // 例如：同一符号表中，'var' 符号和 'class' 符号会冲突。
    // 如果有冲突，报告该问题给该符号的每个声明，然后为该声明创建一个新符号
    //
    // 如果我们创建的新符号既没在符号表中重名也没和现有符号冲突，就将该节点添加为新符号的唯一声明。
    //
    // 否则，就要（将新符号）合并进兼容的现有符号中（例如同一容器中有多个同名的 'var' 时）。这种情况下要把该节点添加到符号的声明列表中。
    symbol = hasProperty(symbolTable, name)
      ? symbolTable[name]
      : (symbolTable[name] = createSymbol(SymbolFlags.None, name));

    if (name && includes & SymbolFlags.Classifiable) {
      classifiableNames[name] = name;
    }

    if (symbol.flags & excludes) {
      if (node.name) {
        node.name.parent = node;
      }

      // 报告每个重复声明的错误位置
      // 报告之前遇到的声明错误
      let message =
        symbol.flags & SymbolFlags.BlockScopedVariable
          ? Diagnostics.Cannot_redeclare_block_scoped_variable_0
          : Diagnostics.Duplicate_identifier_0;
      forEach(symbol.declarations, declaration => {
        file.bindDiagnostics.push(
          createDiagnosticForNode(declaration.name || declaration, message, getDisplayName(declaration))
        );
      });
      file.bindDiagnostics.push(createDiagnosticForNode(node.name || node, message, getDisplayName(node)));
```

```

      symbol = createSymbol(SymbolFlags.None, name);
    }
  } else {
    symbol = createSymbol(SymbolFlags.None, '__missing');
  }

  addDeclarationToSymbol(symbol, node, includes);
  symbol.parent = parent;

  return symbol;
}
```

填充哪个符号表，由此函数的第一个参数决定。例如：添加声明到类型为 `SyntaxKind.ClassDeclaration` 或 `SyntaxKind.ClassExpression` 的容器时，将会调用下面的函数 `declareClassMember` ：

```
function declareClassMember(node: Declaration, symbolFlags: SymbolFlags, symbolExcludes: SymbolFlags) {
  return node.flags & NodeFlags.Static
    ? declareSymbol(container.symbol.exports, container.symbol, node, symbolFlags, symbolExcludes)
    : declareSymbol(container.symbol.members, container.symbol, node, symbolFlags, symbolExcludes);
}
```

绑定器错误报告

绑定错误被添加到源文件的 `bindDiagnostics` 列表中

一个绑定时错误检测的例子是在严格模式下使用 `eval` 或 `arguments` 作为变量名。下面展示了相关的全部代码（多个位置都会调用 `checkStrictModeEvalOrArguments` ，调用栈发自 `bindWorker` ，该函数对不同节点的 `SyntaxKind` 调用不同的检查函数）：

```
function checkStrictModeEvalOrArguments(contextNode: Node, name: Node) {
  if (name && name.kind === SyntaxKind.Identifier) {
    let identifier = <Identifier>name;
    if (isEvalOrArgumentsIdentifier(identifier)) {
      // 首先检查名字是否在类声明或者类表达式中，如果是则给出明确消息，否则报告一般性错误
      let span = getErrorSpanForNode(file, name);
      file.bindDiagnostics.push(
```

```
        createFileDiagnostic(
          file,
          span.start,
          span.length,
          getStrictModeEvalOrArgumentsMessage(contextNode),
          identifier.text
        )
      );
    }
  }
}

function isEvalOrArgumentsIdentifier(node: Node): boolean {
  return (
    node.kind === SyntaxKind.Identifier &&
    ((<Identifier>node).text === 'eval' || (<Identifier>node).text === 'arguments')
  );
}

function getStrictModeEvalOrArgumentsMessage(node: Node) {
  // 向用户提供特定消息，有助他们理解为何会处于严格模式。
  if (getContainingClass(node)) {
    return Diagnostics.Invalid_use_of_0_Class_definitions_are_automatically_in_strict_mode;
  }

  if (file.externalModuleIndicator) {
    return Diagnostics.Invalid_use_of_0_Modules_are_automatically_in_strict_mode;
  }

  return Diagnostics.Invalid_use_of_0_in_strict_mode;
}
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/compiler/binder.html*

检查器

如前所述，检查器使得 TypeScript 更独特，比其它 *JavaScript* 转译器更强大。检查器位于 `checker.ts` 中，当前有 23k 行以上的代码（编译器中最大的部分）

程序对检查器的使用

检查器是由程序初始化，下面是调用栈示意（绑定器一节也展示过）：

```
program.getTypeChecker ->
    ts.createTypeChecker（检查器中）->
        initializeTypeChecker（检查器中） ->
            for each SourceFile `ts.bindSourceFile`（绑定器中）
            // 接着
            for each SourceFile `ts.mergeSymbolTable`（检查器中）
```

与发射器的联系

真正的类型检查会在调用 `getDiagnostics` 时才发生。该函数被调用时（比如由 `Program.emit` 请求），检查器返回一个 `EmitResolver`（由程序调用检查器的 `getEmitResolver` 函数得到），`EmitResolver` 是 `createTypeChecker` 的一个本地函数的集合。介绍发射器时还会再次提到。

下面是该过程直到 `checkSourceFile` 的调用栈（`checkSourceFile` 是 `createTypeChecker` 的一个本地函数）：

```
program.emit ->
    emitWorker (program local) ->
        createTypeChecker.getEmitResolver ->
            // 第一次调用下面的几个 createTypeChecker 的本地函数
            call getDiagnostics ->
                getDiagnosticsWorker ->
                    checkSourceFile

            // 接着
            return resolver
            // 通过对本地函数 createResolver() 的调用，resolver 已在 createTypeChecker 中初始化。
```

全局命名空间合并

`initializeTypeChecker` 中存在以下代码：

```
// 初始化全局符号表（SymbolTable）。
forEach(host.getSourceFiles(), file => {
  if (!isExternalModule(file)) {
    mergeSymbolTable(globals, file.locals);
  }
});
```

基本上是将所有的 `global` 符号合并到 `let globals: SymbolTable = {}` 符号表中（位于 `createTypeChecker` 中）。 `mergeSymbolTable` 主要调用 `mergeSymbol` 函数。

检查器错误报告

检查器使用本地的 `error` 函数报告错误，如下所示：

```
function error(location: Node, message: DiagnosticMessage, arg0?: any, arg1?: any, arg2?: any): void {
  let diagnostic = location
    ? createDiagnosticForNode(location, message, arg0, arg1, arg2)
    : createCompilerDiagnostic(message, arg0, arg1, arg2);
  diagnostics.add(diagnostic);
}
```

> 原文: *https://jkchao.github.io/typescript-book-chinese/compiler/checker.html*

发射器

TypeScript 编译器提供了两个发射器：

- `emitter.ts`：可能是你最感兴趣的发射器，它是 TS -> JavaScript 的发射器
- `declarationEmitter.ts`：这个发射器用于为 *TypeScript* 源文件（`.ts`） 创建声明文件（`.d.ts`）

  本节我们介绍 `emitter.ts`

Promgram 对发射器的使用

Program 提供了一个 `emit` 函数。该函数主要将功能委托给 `emitter.ts` 中的 `emitFiles` 函数。下面是调用栈：

```
Program.emit ->
    `emitWorker` （在 program.ts 中的 createProgram） ->
        `emitFiles` （emitter.ts 中的函数）
```

`emitWorker`（通过 `emitFiles` 参数）给发射器提供一个 `EmitResolver`。`EmitResolver` 由程序的 TypeChecker 提供，基本上它是一个来自 `createChecker` 的本地函数的子集。

发射器函数

emitFiles

定义在 `emitter.ts` 中，下面是该函数的签名：

```
// targetSourceFile 当用户想发射项目中的某个文件时指定，保存时编译（compileOnSave）功能使用此参数
export function emitFiles(resolver: EmitResolver, host: EmitHost, targetSourceFile?: SourceFile): EmitResult {
```

`EmitHost` 是 `CompilerHost` 的简化版（运行时，很多用例实际上都是 `CompilerHost`）

`emitFiles` 中的最有趣的调用栈如下所示：

```
emitFiles ->
    emitFile(jsFilePath, targetSourceFile) ->
        emitJavaScript(jsFilePath, targetSourceFile);
```

emitJavaScript

该函数有良好的注释，我们下面给出它：

```
function emitJavaScript(jsFilePath: string, root?: SourceFile) {
  let writer = createTextWriter(newLine);
  let write = writer.write;
  let writeTextOfNode = writer.writeTextOfNode;
  let writeLine = writer.writeLine;
  let increaseIndent = writer.increaseIndent;
  let decreaseIndent = writer.decreaseIndent;

  let currentSourceFile: SourceFile;
  // 导出器函数的名称，如果文件是个系统外部模块的话
  // System.register([...], function (<exporter>) {...})
  // System 模块中的导出像这样：
  // export var x; ... x = 1
  // =>
  // var x;... exporter("x", x = 1)
  let exportFunctionForFile: string;

  let generatedNameSet: Map<string> = {};
  let nodeToGeneratedName: string[] = [];
  let computedPropertyNamesToGeneratedNames: string[];

  let extendsEmitted = false;
  let decorateEmitted = false;
  let paramEmitted = false;
  let awaiterEmitted = false;
  let tempFlags = 0;
  let tempVariables: Identifier[];
  let tempParameters: Identifier[];
  let externalImports: (ImportDeclaration | ImportEqualsDeclaration | ExportDeclaration)[];
  let exportSpecifiers: Map<ExportSpecifier[]>;
  let exportEquals: ExportAssignment;
  let hasExportStars: boolean;

  /** 将发射输出写入磁盘 */
  let writeEmittedFiles = writeJavaScriptFile;

```

```
  let detachedCommentsInfo: { nodePos: number; detachedCommentEndPos: number }[];

  let writeComment = writeCommentRange;

  /** 发射一个节点 */
  let emit = emitNodeWithoutSourceMap;

  /** 在发射节点前调用 */
  let emitStart = function(node: Node) {};

  /** 发射结点完成后调用 */
  let emitEnd = function(node: Node) {};

  /** 从 startPos 位置开始，为指定的 token 发射文本。默认写入的文本由 tokenKind 提供，
   * 但是如果提供了可选的 emitFn 回调，将使用该回调来代替默认方式发射文本。
   * @param tokenKind 要搜索并发射的 token 的类别
   * @param startPos 源码中搜索 token 的起始位置
   * @param emitFn 如果给出，会被调用来进行文本的发射。
   */
  let emitToken = emitTokenText;

  /** 该函数因为节点，会在发射的代码中于函数或类中启用词法作用域前调用
   * @param scopeDeclaration 启动词法作用域的节点
   * @param scopeName 可选的作用域的名称，默认从节点声明中推导
   */
  let scopeEmitStart = function(scopeDeclaration: Node, scopeName?: string) {};

  /** 出了作用域后调用 */
  let scopeEmitEnd = function() {};

  /** 会被编码的 Sourcemap 数据 */
  let sourceMapData: SourceMapData;

  if (compilerOptions.sourceMap || compilerOptions.inlineSourceMap) {
    initializeEmitterWithSourceMaps();
  }

  if (root) {
    // 不要直接调用 emit，那样不会设置 currentSourceFile
    emitSourceFile(root);
  } else {
```

```
    forEach(host.getSourceFiles(), sourceFile => {
      if (!isExternalModuleOrDeclarationFile(sourceFile)) {
        emitSourceFile(sourceFile);
      }
    });
  }

  writeLine();
  writeEmittedFiles(writer.getText(), /*writeByteOrderMark*/ compilerOptions.emitBOM);
  return;

  /// 一批本地函数
}
```

它主要设置了一批本地变量和函数（这些函数构成 `emitter.ts` 的大部分内容），接着交给本地函数 `emitSourceFile` 发射文本。 `emitSourceFile` 函数设置 `currentSourceFile` 然后交给本地函数 `emit` 去处理。

```
function emitSourceFile(sourceFile: SourceFile): void {
  currentSourceFile = sourceFile;
  exportFunctionForFile = undefined;
  emit(sourceFile);
}
```

`emit` 函数处理 注释 和 实际 *JavaScript* 的发射。实际 *JavaScript* 的发射是 emitJavaScriptWorker 函数的工作。

emitJavaScriptWorker

完整的函数：

```
function emitJavaScriptWorker(node: Node) {
  // 检查节点是否可以忽略 ScriptTarget 发射
  switch (node.kind) {
    case SyntaxKind.Identifier:
      return emitIdentifier(<Identifier>node);
    case SyntaxKind.Parameter:
      return emitParameter(<ParameterDeclaration>node);
    case SyntaxKind.MethodDeclaration:
    case SyntaxKind.MethodSignature:
      return emitMethod(<MethodDeclaration>node);
```

```
      case SyntaxKind.GetAccessor:
      case SyntaxKind.SetAccessor:
        return emitAccessor(<AccessorDeclaration>node);
      case SyntaxKind.ThisKeyword:
        return emitThis(node);
      case SyntaxKind.SuperKeyword:
        return emitSuper(node);
      case SyntaxKind.NullKeyword:
        return write('null');
      case SyntaxKind.TrueKeyword:
        return write('true');
      case SyntaxKind.FalseKeyword:
        return write('false');
      case SyntaxKind.NumericLiteral:
      case SyntaxKind.StringLiteral:
      case SyntaxKind.RegularExpressionLiteral:
      case SyntaxKind.NoSubstitutionTemplateLiteral:
      case SyntaxKind.TemplateHead:
      case SyntaxKind.TemplateMiddle:
      case SyntaxKind.TemplateTail:
        return emitLiteral(<LiteralExpression>node);
      case SyntaxKind.TemplateExpression:
        return emitTemplateExpression(<TemplateExpression>node);
      case SyntaxKind.TemplateSpan:
        return emitTemplateSpan(<TemplateSpan>node);
      case SyntaxKind.JsxElement:
      case SyntaxKind.JsxSelfClosingElement:
        return emitJsxElement(<JsxElement | JsxSelfClosingElement>node);
      case SyntaxKind.JsxText:
        return emitJsxText(<JsxText>node);
      case SyntaxKind.JsxExpression:
        return emitJsxExpression(<JsxExpression>node);
      case SyntaxKind.QualifiedName:
        return emitQualifiedName(<QualifiedName>node);
      case SyntaxKind.ObjectBindingPattern:
        return emitObjectBindingPattern(<BindingPattern>node);
      case SyntaxKind.ArrayBindingPattern:
        return emitArrayBindingPattern(<BindingPattern>node);
      case SyntaxKind.BindingElement:
        return emitBindingElement(<BindingElement>node);
      case SyntaxKind.ArrayLiteralExpression:
        return emitArrayLiteral(<ArrayLiteralExpression>node);
```

```
        case SyntaxKind.ObjectLiteralExpression:
          return emitObjectLiteral(<ObjectLiteralExpression>node);
        case SyntaxKind.PropertyAssignment:
          return emitPropertyAssignment(<PropertyDeclaration>node);
        case SyntaxKind.ShorthandPropertyAssignment:
          return
    emitShorthandPropertyAssignment(<ShorthandPropertyAssignment>node);
        case SyntaxKind.ComputedPropertyName:
          return emitComputedPropertyName(<ComputedPropertyName>node);
        case SyntaxKind.PropertyAccessExpression:
          return emitPropertyAccess(<PropertyAccessExpression>node);
        case SyntaxKind.ElementAccessExpression:
          return emitIndexedAccess(<ElementAccessExpression>node);
        case SyntaxKind.CallExpression:
          return emitCallExpression(<CallExpression>node);
        case SyntaxKind.NewExpression:
          return emitNewExpression(<NewExpression>node);
        case SyntaxKind.TaggedTemplateExpression:
          return emitTaggedTemplateExpression(<TaggedTemplateExpression>node);
        case SyntaxKind.TypeAssertionExpression:
          return emit((<TypeAssertion>node).expression);
        case SyntaxKind.AsExpression:
          return emit((<AsExpression>node).expression);
        case SyntaxKind.ParenthesizedExpression:
          return emitParenExpression(<ParenthesizedExpression>node);
        case SyntaxKind.FunctionDeclaration:
        case SyntaxKind.FunctionExpression:
        case SyntaxKind.ArrowFunction:
          return emitFunctionDeclaration(<FunctionLikeDeclaration>node);
        case SyntaxKind.DeleteExpression:
          return emitDeleteExpression(<DeleteExpression>node);
        case SyntaxKind.TypeOfExpression:
          return emitTypeOfExpression(<TypeOfExpression>node);
        case SyntaxKind.VoidExpression:
          return emitVoidExpression(<VoidExpression>node);
        case SyntaxKind.AwaitExpression:
          return emitAwaitExpression(<AwaitExpression>node);
        case SyntaxKind.PrefixUnaryExpression:
          return emitPrefixUnaryExpression(<PrefixUnaryExpression>node);
        case SyntaxKind.PostfixUnaryExpression:
          return emitPostfixUnaryExpression(<PostfixUnaryExpression>node);
        case SyntaxKind.BinaryExpression:
```

```
            return emitBinaryExpression(<BinaryExpression>node);
          case SyntaxKind.ConditionalExpression:
            return emitConditionalExpression(<ConditionalExpression>node);
          case SyntaxKind.SpreadElementExpression:
            return emitSpreadElementExpression(<SpreadElementExpression>node);
          case SyntaxKind.YieldExpression:
            return emitYieldExpression(<YieldExpression>node);
          case SyntaxKind.OmittedExpression:
            return;
          case SyntaxKind.Block:
          case SyntaxKind.ModuleBlock:
            return emitBlock(<Block>node);
          case SyntaxKind.VariableStatement:
            return emitVariableStatement(<VariableStatement>node);
          case SyntaxKind.EmptyStatement:
            return write(';');
          case SyntaxKind.ExpressionStatement:
            return emitExpressionStatement(<ExpressionStatement>node);
          case SyntaxKind.IfStatement:
            return emitIfStatement(<IfStatement>node);
          case SyntaxKind.DoStatement:
            return emitDoStatement(<DoStatement>node);
          case SyntaxKind.WhileStatement:
            return emitWhileStatement(<WhileStatement>node);
          case SyntaxKind.ForStatement:
            return emitForStatement(<ForStatement>node);
          case SyntaxKind.ForOfStatement:
          case SyntaxKind.ForInStatement:
            return emitForInOrForOfStatement(<ForInStatement>node);
          case SyntaxKind.ContinueStatement:
          case SyntaxKind.BreakStatement:
            return emitBreakOrContinueStatement(<BreakOrContinueStatement>node);
          case SyntaxKind.ReturnStatement:
            return emitReturnStatement(<ReturnStatement>node);
          case SyntaxKind.WithStatement:
            return emitWithStatement(<WithStatement>node);
          case SyntaxKind.SwitchStatement:
            return emitSwitchStatement(<SwitchStatement>node);
          case SyntaxKind.CaseClause:
          case SyntaxKind.DefaultClause:
            return emitCaseOrDefaultClause(<CaseOrDefaultClause>node);
          case SyntaxKind.LabeledStatement:
```

```
            return emitLabelledStatement(<LabeledStatement>node);
        case SyntaxKind.ThrowStatement:
            return emitThrowStatement(<ThrowStatement>node);
        case SyntaxKind.TryStatement:
            return emitTryStatement(<TryStatement>node);
        case SyntaxKind.CatchClause:
            return emitCatchClause(<CatchClause>node);
        case SyntaxKind.DebuggerStatement:
            return emitDebuggerStatement(node);
        case SyntaxKind.VariableDeclaration:
            return emitVariableDeclaration(<VariableDeclaration>node);
        case SyntaxKind.ClassExpression:
            return emitClassExpression(<ClassExpression>node);
        case SyntaxKind.ClassDeclaration:
            return emitClassDeclaration(<ClassDeclaration>node);
        case SyntaxKind.InterfaceDeclaration:
            return emitInterfaceDeclaration(<InterfaceDeclaration>node);
        case SyntaxKind.EnumDeclaration:
            return emitEnumDeclaration(<EnumDeclaration>node);
        case SyntaxKind.EnumMember:
            return emitEnumMember(<EnumMember>node);
        case SyntaxKind.ModuleDeclaration:
            return emitModuleDeclaration(<ModuleDeclaration>node);
        case SyntaxKind.ImportDeclaration:
            return emitImportDeclaration(<ImportDeclaration>node);
        case SyntaxKind.ImportEqualsDeclaration:
            return emitImportEqualsDeclaration(<ImportEqualsDeclaration>node);
        case SyntaxKind.ExportDeclaration:
            return emitExportDeclaration(<ExportDeclaration>node);
        case SyntaxKind.ExportAssignment:
            return emitExportAssignment(<ExportAssignment>node);
        case SyntaxKind.SourceFile:
            return emitSourceFileNode(<SourceFile>node);
    }
}
```

通过简单地调用相应的 `emitXXX` 函数来完成递归，例如 `emitFunctionDeclaration`

```
function emitFunctionDeclaration(node: FunctionLikeDeclaration) {
  if (nodeIsMissing(node.body)) {
    return emitOnlyPinnedOrTripleSlashComments(node);
  }
```

```

  if (node.kind !== SyntaxKind.MethodDeclaration && node.kind !==
SyntaxKind.MethodSignature) {
    // 会把注释当做方法声明的一部分去发射。
    emitLeadingComments(node);
  }

  // 目标为 es6 之前时，使用 function 关键字来发射类函数（functions-like）声明，包括箭头函数
  // 目标为 es6 时，可以发射原生的 ES6 箭头函数，并使用宽箭头代替 function 关键字.
  if (!shouldEmitAsArrowFunction(node)) {
    if (isES6ExportedDeclaration(node)) {
      write('export ');
      if (node.flags & NodeFlags.Default) {
        write('default ');
      }
    }

    write('function');
    if (languageVersion >= ScriptTarget.ES6 && node.asteriskToken) {
      write('*');
    }
    write(' ');
  }

  if (shouldEmitFunctionName(node)) {
    emitDeclarationName(node);
  }

  emitSignatureAndBody(node);
  if (
    languageVersion < ScriptTarget.ES6 &&
    node.kind === SyntaxKind.FunctionDeclaration &&
    node.parent === currentSourceFile &&
    node.name
  ) {
    emitExportMemberAssignments((<FunctionDeclaration>node).name);
  }
  if (node.kind !== SyntaxKind.MethodDeclaration && node.kind !==
SyntaxKind.MethodSignature) {
    emitTrailingComments(node);
  }
```

```
}
```

发射器源映射（SourceMaps）

如前所述 `emitter.ts` 的大部分是本地函数 `emitJavaScript`（我们之前展示过该函数的初始化例程）。它主要是设置一批本地变量并交给 `emitSourceFile` 处理。下面我们再看一遍这个函数，这次我们重点关注 `SourceMap` 的部分：

```
function emitJavaScript(jsFilePath: string, root?: SourceFile) {

    // 无关代码 ........... 已移除
    let writeComment = writeCommentRange;

    /** 将发射的输出写到磁盘上 */
    let writeEmittedFiles = writeJavaScriptFile;

    /** 发射一个节点 */
    let emit = emitNodeWithoutSourceMap;

    /** 节点发射前调用 */
    let emitStart = function (node: Node) { };

    /** 节点发射完成后调用 */
    let emitEnd = function (node: Node) { };

    /** 从 startPos 位置开始，为指定的 token 发射文本。默认写入的文本由 tokenKind 提供，
      * 但是如果提供了可选的 emitFn 回调，将使用该回调来代替默认方式发射文本。
      * @param tokenKind 要搜索并发射的 token 的类别
      * @param startPos 源码中搜索 token 的起始位置
      * @param emitFn 如果给出，会被调用来进行文本的发射。*/
    let emitToken = emitTokenText;

    /** 该函数因为节点，会在发射的代码中于函数或类中启用词法作用域前调用
      * @param scopeDeclaration 启动词法作用域的节点
      * @param scopeName 可选的作用域的名称，默认从节点声明中推导
      */
    let scopeEmitStart = function(scopeDeclaration: Node, scopeName?: string) { };

    /** 出了作用域后调用 */
```

```
    let scopeEmitEnd = function() { };

    /** 会被编码的 Sourcemap 数据 */
    let sourceMapData: SourceMapData;

    if (compilerOptions.sourceMap || compilerOptions.inlineSourceMap) {
        initializeEmitterWithSourceMaps();
    }

    if (root) {
        // 不要直接调用 emit，那样不会设置 currentSourceFile
        emitSourceFile(root);
    }
    else {
        forEach(host.getSourceFiles(), sourceFile => {
            if (!isExternalModuleOrDeclarationFile(sourceFile)) {
                emitSourceFile(sourceFile);
            }
        });
    }

    writeLine();
    writeEmittedFiles(writer.getText(), /*writeByteOrderMark*/ compilerOptions.emitBOM);
    return;
```

重要的函数调用是 `initializeEmitterWithSourceMaps`，该函数是 `emitJavaScript` 的本地函数，它覆盖了部分已定义的本地函数。覆盖的函数可以在 `initalizeEmitterWithSourceMap` 的底部找到：

```
// `initializeEmitterWithSourceMaps` 函数的最后部分

writeEmittedFiles = writeJavaScriptAndSourceMapFile;
emit = emitNodeWithSourceMap;
emitStart = recordEmitNodeStartSpan;
emitEnd = recordEmitNodeEndSpan;
emitToken = writeTextWithSpanRecord;
scopeEmitStart = recordScopeNameOfNode;
scopeEmitEnd = recordScopeNameEnd;
writeComment = writeCommentRangeWithMap;
```

就是说大部分的发射器代码不关心 `SourceMap`，它们以相同的方式使用这些（带或不带

SourceMap 的）本地函数。

> 原文: *https://jkchao.github.io/typescript-book-chinese/compiler/emitter.html*